Sarah Sayeed Qureshi

AI Planning Methods for Subsea IMR Operations

Masteroppgave i Kybernetikk og Robotikk
Veileder: Anastasios Lekkas
Juli 2020

Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

NTNU
Norwegian University of
Science and Technology

Summary

Inspection, Maintenance and Repair (IMR) operations are essential in the subsea industry, due to the high number of subsea installations. The current approach of performing these operations are by human-operated underwater vehicles with a low level of autonomy. Consequently, this thesis is exploring new methods for increasing the level of autonomy by introducing an aspect of deliberative planning with the use of Automated Planning and Scheduling (AI Planning). The use of AI Planning is inspired from space applications, as these have shown great progress in increasing the level of autonomy for mission planning problems.

This thesis also delves into a novel approach of using Reinforcement Learning (RL) in order to solve a mission planning problem in the domain of subsea IMR operations. The explored methods for solving the defined mission planning problem are Hierarchical Task Network (HTN) and Graphplan from the domain of AI Planning and Q-learning from Reinforcement Learning.

Both the AI Planning methods and the RL method are used to solve a mission planning problem which has been developed with important feedback from Remotely Operated Vehicle (ROV)-operators from the subsea industry. In terms of the mission planning problem, the aim has been to model and develop a mission in such a manner that it would reflect real IMR operations. Hence, the domain of the mission planning problem contains elements such as a subsea panel, a warehouse and a docking station which are common parts of subsea installations.

The results obtained from the three methods, i.e. HTN, Graphplan and Q-learning give an indication of how a software agent can solve a mission planning problem in the domain of subsea IMR operations. Furthermore, the methods are compared to one another in their ability to solve the mission planning problem both with and without a replanning aspect. The two AI Planning methods are principally compared on their total runtime when solving the mission, whilst the Q-learning algorithm is evaluated on the ability of actually solving the mission planning problem by exploring and learning about the environment.

Sammendrag

Inspisering-, vedlikehold og reparasjonsoppdrag (eng: Inspection, Maintenance and Repair) er helt essensielle i subsea-industrien, grunnet det store antallet installasjoner som finnes på havbunnen. Nåværende tilnærming til disse oppdragene er bruken av menneskestyrte undervannsdroner som i svært liten grad er autonome. Av den grunn utforsker denne avhandlingen nye metoder for å kunne øke nivået av autonomi i disse oppdragene ved å benytte seg av "AI Planning". Inspirasjonen til å benytte seg av nettopp disse metodene kommer fra romfartsindustrien hvor kunstig intelligent planlegging (eng: AI Planning) har hatt veldig lovende fremgang i å øke nivået av autonomi i et oppdrag.

Denne avhandlingen dykker også inn i en ny tilnærming av å løse planleggingsproblemer for inspisering-, vedlikehold og reparasjonsoppdrag ved bruk av forsterkende læring (eng: Reinforcement Learning). The utforskede metodene inkluderer "Graphplan" og Hierarchical Task Network" (HTN) fra kunstig intelligent planlegging og Q-læring fra forsterkende læring.

Både metodene innenfor kunstig intelligent planlegging og forsterkende læring er brukt for å løse et planleggingsoppdrag som har blitt utviklet i samarbeid med fjernstyrte undervannsfarkostpiloter (eng: Remotely Operated Vehicle-operators) fra subseaindustrien. Hovedmålet er å modellere og utvikle et planleggingsoppdrag som skal kunne gjenspeile et reelt oppdrag i industrien. Dermed består omgivelsene i det utviklede domenet av elementer som en dockingstasjon, et varehus og et panel med et gitt antall ventiler, siden dette er vanlige deler på en subsea-installasjon.

De oppnådde resultatene fra de tre implementerte metodene, dvs. HTN, Graphplan og Q-læring, gir en indikasjon på hvordan en såkalt intelligent agent kan løse et slikt oppdrag som er i sammenheng med inspisering-, vedlikehold og reparasjon av subsea-installasjoner. I tillegg er methodene sammenlignet med hverandre basert på deres evne til å løse et slikt oppdrag både med og uten omplanlegging. De to metodene innenfor kunstig intelligent planlegging sammenlignes basert på kjøretiden deres når de løser planlegginsoppdraget, mens Q-læring evalueres basert på dens evne til å løse det forenklede scenarioet ved å utforske og lære av sine omgivelser.

Preface

This thesis represents the conducted work of a master's project affiliated with the department of Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU). The works of this thesis has been conducted during spring 2020 and focuses on how to model a Inspection, Maintenance and Repair mission and thereby solve it using Automated planning and scheduling (AI Planning) and Reinforcement Learning. The works of this thesis is a continuation of the project thesis from fall 2019 [1], where three AI Planning methods were explored. Additionally, this project is affiliated with the work conducted on "AI Planning for Underwater Intervention Drone" [2] by Libo Xue at the department of Engineering Cybernetics.

It is worth mentioning that this current master's project was initially about the potential use of NASA's planner EUROPA to solve the mission planning problem in the domain of an IMR operation. Furthermore, the planner and the solution would be connected to the low-level controller T-REX. Due to the unfortunate outbreak of Covid-19 and closing of the university, the initial plan of collaborating closely together in the implementation of EUROPA and T-REX was no longer possible. The objective of the master's thesis was therefore completely modified.

The development of this master's thesis would not have been possible without the help and support of professor Anastasios Lekkas at the department of Engineering Cybernetics. He has helped formulate a clear objective and problem definition for the thesis, which has been both relevant and interesting. Professor Lekkas also assisted in re-formulating the objective of the thesis after the impact of the Covid-19 pandemic. Furthermore, Libo Xue, offered great advice when modeling the IMR operation as a mission planning problem and in selecting the most appropriate AI planning methods for solving it.

As the work is a continuation of the project thesis, there are several aspects which have been re-used from the project report. Among these are some parts of the background information in Chapter 1, the theory regarding AI Planning in Chapter 2 and the explanation of the Planning Domain Definition Language (PDDL) in Chapter 3. However, all the parts from the project thesis have been modified to match the works of this thesis, in addition to being improved and elaborated further.

The two AI planning methods and the Reinforcement Learning method are all implemented in Python 3 and details about the libraries can be found in Chapter 3. The libraries used for HTN, Graphplan and Q-learning are retrieved from [3], [4] and [5], respectively and are all Python 3 compatible.

Finally, draw.io is used to produce illustrations related to the planning problems. In addition, illustrations from other resources are also reproduced in the same software if necessary. The reproduced illustrations are still credited to the original reference.

Acknowledgment

First and foremost, I would like to thank my supervisor Anastasios Lekkas for all the guidance, motivation and interesting discussion we have had throughout the period of the master project. Furthermore, I would also like to thank Libo Xue for giving me good advice regarding the implementation of the AI Planning methods.

I would additionally also thank my fellow graduate student and friends who have supported and helped me throughout this period of time, with both inspiring discussions and motivational pep-talks. An extra thank you goes out to my dear friend Misbah, who has taken out time to help me proofread this thesis.

Last, but not least I would like to thank my family for the immense support they have been during my time at NTNU, especially this last semester with the unfortunate impact of the Covid-19 pandemic. I am very thankful for my parents and brother, Sami, who drove all the way to Trondheim in order to bring me back to Bergen so that I could finish my degree surrounded by family. I am extremely grateful for my sister, Sophia, who has comforted and supported me throughout all my years at NTNU, in addition to take out time to help me with my master thesis.

01.07.2020
Sarah Sayeed Qureshi

Table of Contents

List of Tables

List of Figures

List of Algorithms

Listings

Abbreviations

AI	=	Artificial Intelligence
API	=	Application Programming Interface
AUV	=	Autonomous Underwater Vehicle
FOL	=	First-Order Logic
GUI	=	Graphic User Interface
HTN	=	Hierarchical Task Network
IMR	=	Inspection, Maintenance and Repair
MDP	=	Markov Decision Process
ML	=	Machine Learning
RL	=	Reinforcement Learning
ROV	=	Remotely Operated Vehicles
SARSA	=	State-Action-Reward-State-Action
SHOP	=	Simple Hierarchical Ordered Planner
TD	=	Temporal-Difference
UID	=	Underwater Intervention Drone
UUV	=	Unmanned Underwater Vehicle

Chapter 1

Introduction

This chapter gives some introductory notions on the background of conducting the work in this thesis. The chapter also describes projects and work already conducted in the domain of underwater vehicles.

1.1 Background and Motivation

There exists more than 5000 subsea wells on the Norwegian Continental Shelf that require continuous inspection and repair due to aging equipment. Inspection, maintenance and repair (IMR) operations on these subsea installations currently require several expensive resources, such as offshore support vessels with experienced operators on-board, in addition to Remotely Operated Vehicles (ROV). It is necessary to have these aforementioned resources available simultaneously in order to perform IMR operations. This might not always be the case when a sudden need of repair or inspection occurs, resulting in a higher response time. Failure on a critical part will consequently result in an overall increased downtime for the installation [6]. A step towards reducing the overall cost and response time of IMR operations is to increase the level of autonomy in these operations. Consequently, this has led to a larger focus on research and development of Autonomous Underwater Vehicles (AUV) which could potentially replace the aforementioned de facto standard of using ROVs [7].

An important aspect of underwater vehicles which perform IMR operations are their abilities of being persistent, especially when autonomous. This is also called persistent autonomy, and describes the ability of a vehicle to be autonomous for an extended duration of time, in addition to performing complex tasks without human intervention [7]. Persistent autonomy is especially important in IMR operations, as the working conditions for the underwater vehicles are quite challenging on the seabed [8]. However, it is important to clarify that the first step towards persistent autonomy is to develop a well-defined and robust autonomous vehicle which can perform the IMR operations. Consequently, the desire

of achieving persistent autonomy for IMR operation is a long-term goal.

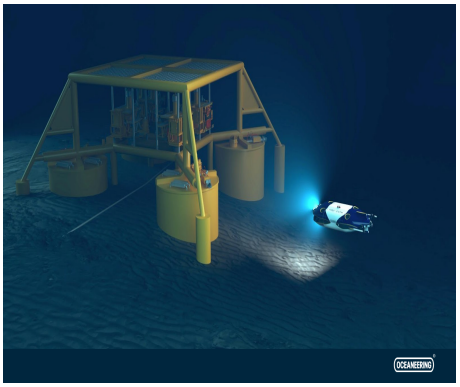
The development of autonomy in subsea operations and underwater vehicles can draw inspiration from space applications, as this is a domain where the development of autonomy has progressed further. A main point which has been demonstrated in the development of autonomy within space applications, is that planning is a critical aspect of it [9]. Planning is defined as an abstract, explicit deliberation process where the actions are both chosen and organized based on anticipating the expected outcome. A plan also takes into consideration the state of mission, state of vehicle and potential fallbacks [10]. Furthermore, in space applications it has been demonstrated that autonomy requires several modules which need to collaborate together, including control, navigation and localization systems. This is also the case for planning in the subsea domain, in which numerous projects conducted by the CIRS lab at the University of Girona, Spain have given valuable results for localization approaches [11], underwater intervention [12] and implementation of a control architecture for the lab's AUV, Girona 500 AUV [13].

However, it is worth mentioning that several other institutions have also had valuable contributions in developing essential modules for increasing the level of autonomy in underwater vehicles. Among others, the Norwegian Defence Research Establishment (FFI) has made progress in developing a navigation system for the HUGIN AUV [14] while the Monterey Bay Aquarium Research Institute (MBARI) proposed a deliberative control architecture for the MBARI AUV [15].

A project which made some of the first steps in incorporating the necessary aforementioned systems for an autonomous vehicle, was the NextGen IMR project, conducted between 2014-2017. This project's approach was inspired from the slogan "From outer space to ocean space", which reflects the inspiration gained by space applications for solving mission planning problems [16]. The PANDORA project was conducted prior, i.e. between 2012-2015, and focused on developing persistent autonomous underwater robots. The main goal was to decrease the level of operator interventions when executing tasks, while also increasing the overall complexity of the tasks [17]. Both projects made important progress in developing underwater path planning systems, but still the important aspect of autonomy; planning, is yet to be explored within the subsea domain.

In addition to the aforementioned projects, the subsea industry has also demonstrated promising development of underwater vehicles. For instance, during SPE Offshore Europe 2019, a seminar and conference which is arranged every other year, Oceaneering showcased their new resident underwater vehicle "Freedom". This underwater vehicle can function both as an ROV, when tethered, and an AUV when untethered [18]. A different type of resident underwater robot is developed by the Norwegian company Eelume, which has quite an innovative design as it is shaped as a snake. This increases the vehicle's flexibility and it can access spaces which are difficult for other vehicles with the same goal [19]. One of the largest operator in the offshore industry is Equinor, which is collaborating with both Oceaneering and Eelume to develop Underwater Intervention Drones. An Underwater Intervention Drone (UID) is a concept introduced by Equinor and represents

a hybrid of Remotely Operated Vehicles and Autonomous Vehicles [20].



(a) Oceaneering's new residing AUV, Freedom [21]

(b) Eelume's resident snake robot [19]

Figure 1.1: Examples of vehicles which can be used for IMR operations

The current progess in developing autonomous underwater vehicles is that the existing mission control systems normally plan a priori with either minimal or no flexibility to implementing changes during execution. This results in an overall lower level of autonomy [22]. An approach used in space applications to increase a system's autonomy has been the use of automated planning and scheduling technology, also called AI Planning [23]. The same article also explains how the use of AI Planning benefits the system it is applied to. Some examples are the increase of responsiveness, interactivity and productivity, in addition to the reduction of cost. These are all desired factors within IMR operations, seen from an industrial perspective [6].

Another potential approach for solving planning problems based on a mission control system is by using learning-based methods [24]. It has been shown in several cases that many planners in automated planning fail to scale-up from a toy problem to a real scenario and to result in good solutions. In order to overcome this inconvenience of classical planners in automated planning, the learning-based planner would exploit domain-specific knowledge in order to result in good solutions [25].

1.2 Objective

This master thesis is a continuation of the work carried out in the project thesis [1]. The project thesis explored several methods within automated planning and scheduling for solving a mission planning problem reflecting IMR operations on subsea installations. The work is targeted towards the oil and gas industry, as they are in charge of majority of the subsea installations. Hence, the defined mission planning problem in Chapter 4 has been formulated by gaining important industrial input from the ROV operators Jon Englund and Peter Baastad at Oceaneering.

This thesis will use two of the explored AI planning methods; Hierarchical Task Network (HTN) and Graphplan in order to solve a new mission planning problem which would reflect an IMR operation. However, the formulated mission in this thesis has a higher level of complexity compared to previously. The increased mission complexity will reflect a real scenario to a greater extent. Additionally, this thesis will explore how the aforementioned classical AI planning methods would create a plan for solving the defined mission both with and without uncertainty. When the mission contains some uncertain elements, the overall plan might change under execution.

The main objectives of this thesis include the modeling and structuring of an IMR mission, in addition to a comparison and analysis of the rationality of the produced plan. Furthermore, determining the overall efficiency of the planner is also an objective. The rationality of a plan is determined by if the ordered action sequences are correct for reaching the goal of the mission, while the efficiency is defined based on two different aspects. These aspects are the time required by the planner to produce a plan and the time required by the planner for re-planning purposes due to uncertainties.

Moreover, the aspect of using a learning-based planner in order to solve the mission will also be explored. An attempt to solve the formulated mission planning problem will be made with the use of Reinforcement Learning (RL); more precisely the method of Q-learning.

1.3 Contribution

The contribution of this thesis is the formulation and modeling of an IMR mission with the assistance from ROV-operators from the industry. Furthermore, the defined mission planning problem focuses only on the panel and omits the pipeline, making the scenario more specific.

Furthermore, the work of this thesis attempts to solve a simple mission planning problem by the use of Reinforcement Learning. Moreover, a new custom environment is created to represent the subsea domain with all the different installations. This environment can be further developed, and might be an asset when trying other RL algorithms.

Another contribution is the comparison between the AI Planning methods, Hierarchical Task Network and Graphplan. The methods are compared based on their run-time and their ability to solve a mission planning problems which might require replanning.

1.4 Outline

This thesis is divided into seven chapters. Following the current introductory chapter is the theory which reflects the prerequisites needed for gaining a better understanding of both the used AI planning and learning-based methods. Additionally, the theory chapter will also give important insight in terminology used when explaining the implementation and results. Following, the tools and library used for implementing and modelling the mission planning problem are introduced. The chapter will also go into further details of how the used libraries are comprised. Chapter four contains the mission formulation of the IMR operation used in the purpose of this thesis. Additionally, the chapter include the actual implementation of the planners in the context of the mission planning problem. Chapter five contains the obtained results and analysis, and the thesis is concluded in chapter six. Lastly, chapter seven sums up future work that can be performed in continuation of the works of this thesis.

Chapter 2

Theory

This chapter will give the reader an overview of important aspects of automated planning and scheduling, while also introducing one to some machine learning aspects. Moreover, it will present three methods within AI Planning; STRIPS, Hierarchical Task Network and Graphplan, even though only the two latter are used in the works of this thesis. STRIPS is a well-known method within AI Planning and Graphplan is a further development of it. Therefore it is essential to have some knowledge about STRIPS in order to fully understand Graphplan. Thereafter, the chapter explores machine learning aspects and reinforcement learning. Furthermore, this chapter sets the ground for understanding the upcoming chapters about implementation, modeling and the obtained results.

2.1 Automated Planning and Scheduling

Two of the planning methods which are used in the works of this thesis are within the domain of automated planning and scheduling, hereby called AI planning. AI planning is a subarea of Artificial Intelligence (AI) which explores deliberate systems, i.e. systems that require some extent of reasoning in order to perform actions [10]. Such a reasoning process, also called deliberation for acting, is carried out by an artificial agent, which both chooses and performs its actions based on how it can achieve the intended objectives [26].

Deliberation for acting is considered an important aspect of autonomous systems which operate in a diverse environment and execute a wide variety of tasks. These are mainly autonomous systems which require low level of human intervention. It is worth mentioning that there exist numerous autonomous systems which do not require any deliberation, even though they also have a low level of human intervention during operations. The main difference is that the mentioned systems operate in a known environment with predefined, simpler and fewer tasks, and consequently do not need to reason in order to make any decisions [27].

An artificial agent, also called actor, determines which actions to perform and how to perform them based on a predictive model. The predictive model is further divided into a descriptive model and an operational model, where the first describes the set of possible states which can be reached by performing a certain action. The operational model describes how to perform the action, which includes the knowledge of which commands are necessary for executing a certain action.

Furthermore, AI Planning can be divided into two subcategories of planners; state-space and plan-space planners, which will be explored further throughout this thesis. State-space planners are commonly used within AI planning as state-space search algorithms, i.e. algorithms which solve planning problems described in state space and are considered to be the simplest planning algorithms [10].

2.1.1 State-Space Planning

A state space is a way of gathering all available information regarding a planning problem in the same space, where a state-transition system is such an example. It is defined as the following 4-tuple: $\sum = (S, A, \gamma, cost)$ or 3-tuple: $\sum = (S, A, \gamma)$ if the $cost$ is omitted. The different terms can be explained in the following manner: $\sum$ is the general term for a system, S contains the finite number of states in which the system can be, A contains all the finite number of actions which the actor can execute and γ describes the state-transition function. The latter is defined as: $\gamma : S \times A \rightarrow S$, and can be explained as a partial function used to evaluate if an action is applicable in the current state. The cost is also defined as a partial function, which can represent numerous factors such as time, monetary cost or even energy. The aim is normally to minimize this cost. Since a 3-tuple does not have an explicitly defined cost, it is defined as $cost = 1$ whenever $\gamma(s, a)$ is defined [10].

The state space of a planning problem can be decomposed into an initial state, a goal state and a transition model. The state space describes all the reachable states from the initial state given some sequence of actions which connects the initial state and the goal state. Furthermore, the state space also forms a graph, which is a data structure consisting of nodes and links which are connecting the nodes to one another [28]. In the case of state space planning models, the nodes and links in the graph correspond to the states and actions in the state space, respectively. In the state space, a path can be defined as a sequence of states which are connected by a sequence of actions [29].

In order to further simplify the classical AI planning problems, several restrictive assumptions, also called classical planning assumptions, are imposed on the system. In addition to the actions A and states S being a finite number, the environment also has to be finite and static. This means that changes in the systems are only caused by the execution of some action. Secondly, time is not modeled in any explicit way. The actions A and states S are both considered as discrete sequences. The last assumption is that the systems is deterministic, i.e. every possible state caused by the different available actions is predefined. Imposing the mentioned assumptions on a system contributes in eliminating potential errors in the actor's deliberation process. Systems which do not satisfy the aforementioned assumptions could still be acceptable if the errors are neither infrequent nor have severe

consequences [27].

The following two sections will present two methods within state-space planning, STanford Research Institute Problem Solver (STRIPS) and Graphplan. Both methods exploit structuring the available information such as states and actions in a graph.

2.1.1.1 STanford Research Institute Problem Solver

STanford Research Institute Problem Solver, hereby called STRIPS, is a problem solver introduced in order to improve the efficiency of state space planning problems by search space size-reduction [10]. In STRIPS, a planning problem is described as a set of well-formed formulas (wffs), i.e. some finite sequence of symbols, for first-order predicate calculus. First-order predicate calculus is also known as first-order logic (FOL) and is used as a language which assumes that a world can be described as objects with some relation between them which either do or do not hold [29].

The problem space, i.e. the search space for the problem, for STRIPS can be described by three entities. First, an initial world model, which is a set describing the current state of the world, operators described as a set with the actions respective precondition, i.e. conditions which need to be satisfied in the world model and effects on it, and last a goal state which portrays the desired final state [30].

Figure 2.1 illustrates an example from [30] of a simple worlds space in STRIPS-representation. It describes the model for a world where there are two locations, a and b, a robot and two boxes, B and C. The robot is placed at location a and the boxes B and C are both placed at location b. Additionally, three operators are defined. These describe at which location the robot and boxes are with respect to each other.

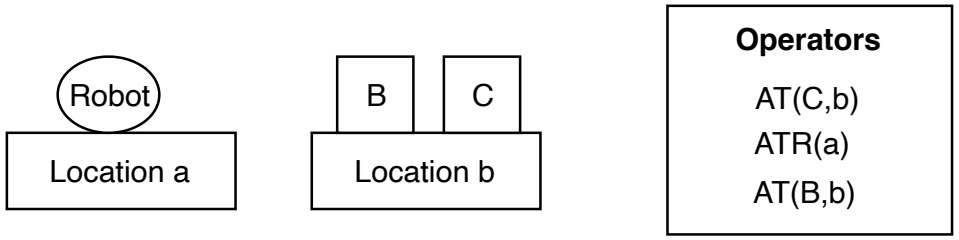


Figure 2.1: Example of a problem space for STRIPS with corresponding operators [30]

In order to solve a planning problem with the use of STRIPS, a search graph has to be created based on the initial state, possible actions and goal state. For a very simple planning problem, i.e. a planning problem with few states and actions operations, the search graph can be created by starting in the initial state and applying all the possible actions. Hence, the initial state will be the starting node of the search tree. The application of actions would result in successor nodes, i.e. the states one can reach by applying some action. For each successor the possible actions are applied, until one of the reached nodes correspond to the goal state. However, for larger world models, i.e. models with many states and action operations, this approach would result in an immense search-tree. STRIPS reduces the search space by using a so-called GPS strategy which extracts the differences between the current state of the world and the goal state and identifies operators which could reduce these differences. Upon detection of such an operator the STRIPS method attempts to produce a world model to which the identified operator can be applied to, which is then considered as a subproblem. Solving the subproblem results in reconsidering the original goal as the desired world model [30].

The STRIPS planner is described in algorithm 1, which has been retrieved from [10]. It is worth mentioning that the described STRIPS algorithms is ground, which consequently means that the states and actions are also ground. This means that neither the states nor the actions contain any free variables, and hence the operators in the planning problem are not generalized [31].

Algorithm 1 takes a planning problem defined with a set of operators O, an initial state s and a goal state g as input. It initializes the plan π as empty and then runs in a loop until it either finds a solution to the planning problem or fails in the attempt. As a starting point it checks if the initial state s_0 satisfies the desired goal state g, as this would terminate the algorithm. Following, a set of actions A are declared based on which actions are relevant for the goal g. As mentioned previously, the STRIPS algorithm starts with the goal state and works its way backtracks to the initial state in order to find a solution and therefore the set of actions A must be related to g. The algorithm continues by non-deterministically choosing an action $a \in A$, and then executing a recursive call of the STRIPS algorithm with the operator O related to the chosen action a, the state s and the sub-goal defined as any of the preconditions of the action a. Thereafter, the algorithm checks if the recursive call actually returns a valid sub-plan. Based on this, the current state s is updated and finally the partial plan π' for solving the sub-goal is added to the plan for the overall solution π.

Algorithm 1 Ground-STRIPS

```
1:  function GROUND-STRIPS(O, s, g)
2:      π ← empty plan
3:      loop
4:          if s satisfies g then return π
5:          A ← {a | a is ground instance of an operator O and a is relevant for g }
6:          if A= ∅ then return failure
7:          non-deterministically choose any action a ∈ A
8:          π′ ← GROUND-STRIPS(O, s, precond(O))
9:          if π′ = failure then return failure
10:         s ← γ(s, π′)                          ▷ π′ achieves precond(a) from s
11:         s ← γ(s, a)                           ▷ s now satisfies precond(a)
12:         π ← π.π′.a
```

The algorithm is applied upon a planning domain consisting of ground atoms, i.e. a predicate with a finite number of real objects. Consequently, the objects can not be variables and have to be pre-defined [29]. Operators in the STRIPS domain can therefore be considered as functions requiring parameters, which are passed on as objects.

One of the key points of the STRIPS method is how it structures the available actions and states into a graph. In order to find a solution to a certain planning problem, any graph-search algorithm can be used. Examples of such graph-searching algorithms are breadth-first, depth-first and A* (A-star). It is worth mentioning that the latter is commonly used when using the STRIPS method [29].

2.1.1.2 Sussman Anomaly

A disadvantage of state-space planners, and thereby also STRIPS, is that they can suffer from the Sussman Anomaly. The Sussman anomaly can occur for a planning problem where the are multiple goals or subgoals which need to be achieved at the same time. The difficulty occurs when for instance the STRIPS planner achieves one subgoal and has yet to achieve another subgoal in order to solve the planning problem. The problem is when the effects of achieving one goal undoes the already existed goal, because in this way the final solution might not be found unless some re-planning is initiated [10].

The reason that the Sussman anomaly can occur in STRIPS planning problems are that the search space contains infinitely many solution, but they are redundant. Therefore, a possible way of finding a solution is to solve for each of the subgoals before combining the found solutions. This finding of solution for the Sussman Anomaly contributed in the development of plan-space planning techniques, which searches for solutions through a space of partial-order plans rather than states. Plan-space planning will be further explored later in this chapter.

A classical example which illustrates the occurrence of the Sussman anomaly, is the dock-worker-robot problem illustrated in figure 2.2 with the defined initial and goal states as

also elaborated in listing 2.1. The domain of problem include one crane, one location, three containers $(c1, c2, c3)$ and five piles $(p1, p2, q1, q2, q3)$. A possible solution, which is also the shortest, is that the crane picks up container $c3$ from the pile $p1$ and puts it in pile $q1$. Furthermore, the crane picks up container $c1$ and and puts it in pile $p2$ on top of container $c2$ as this satisfies the first part of the goal, i.e. $on(c1, c2)$. The problem is that in order to achieve the second part of the goal, i.e. $on(c2, c3)$, the planner has to undo one of the already achieved sub-goals in order to achieve the other sub-goal which is initially not intuitive for the planner.

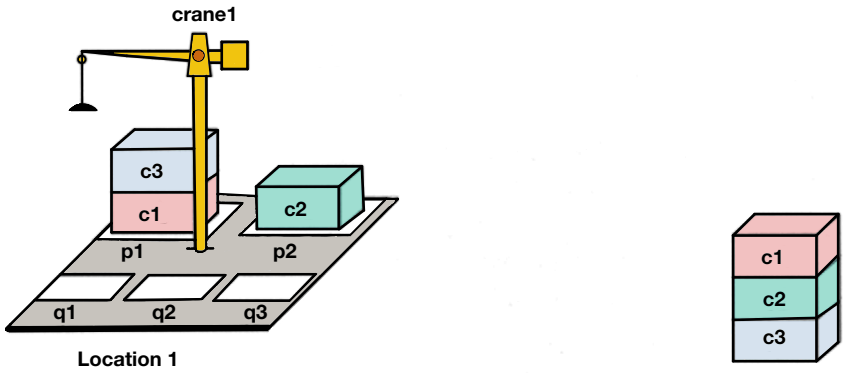


Figure 2.2: Example of a STRIPS problem with initial and goal state suffering from the Sussman anomaly [10]

```
init:
    in(c3, p1) and top(c3, p1) and on(c3, c1) and
    on(c1,pallet) and in(c2, p2) and top(c2, p2) and
    on(c2, pallet) and top(pallet, q1) and top(pallet, q2) and
    top(pallet, q3) and and empty(crane1)

goal:
    on(c1, c2) and on(c2, c3)
```

Listing 2.1: Initial and goal state for the STRIPS problem suffering from the Sussman anomaly

2.1.1.3 Graphplan

Graphplan is a planning method within classical AI Planning which exploits a data structure called planning graph. It is commonly used in STRIPS-resembling domains and always returns the shortest partial-order plan [32]. A planning graph data structure always return an admissible heuristic as it either reports that the goal is not reachable from the current state or it estimates the number of steps necessary in order to reach the goal. The estimate is always lower than the actual number of steps, which satisfies the third point for an admissible heuristic as stated above for the heuristic in A* search [29].

A planning graph is defined as a directed graph which is organized into numerous levels, alternating between a level of states S and actions A until the termination condition is satisfied. The first level is S_0 which describes the initial state of the planning problem, then the next level A_0 is describing the actions that are applicable in A_0. This structure is illustrated in figure 2.3 for a simple planning problem. It is worth mentioning that planning graphs can only be used in propositional planning problems, i.e. planning problems with no variables [29].

The Graphplan planner also tackles some of the problems and drawbacks of the STRIPS planner. For instance, a planning graph includes some mutual exclusion (mutex) conditions. Two actions are considered mutex if their effects include changing the same state variable to different values. It is worth reminding that for the STRIPS planner mutex actions resulted in the occurence of the Sussman anomaly. Thus, the set of actions A_i where i represent the level of the planning graph only include the actions a_i whose preconditions are not mutex in S_i [27]. There are also a few examples of mutex relations which may occur for a planning graph [29]:

- **Inconsistent effects:** if two actions change the same variable but to different values. For instance, in the example illustrated in figure 2.3 the action $Eat(cake)$ and the persistence of $\neg Eaten(cake)$ are inconsistent as the first has effect $Eaten(cake)$ and the latter has the effect $\neg Eaten(cake)$.
- **Interference:** the effect of one action is equivalent to the precondition of another action, but negated. For the example in figure 2.3 the persistent action of $\neg Eaten(cake)$ interferes with the action $Eat(cake)$ as the precondition of the latter, i.e. $\neg Eaten(cake)$ is the negated version of the effect of $Eat(cake)$, i.e. $Eaten(cake)$.
- **Competing needs:** one precondition is mutex with another precondition. For instance, for the example in figure 2.3 $Eat(cake)$ and $Bake(cake)$ in A_1 are mutex since they are both depend on the value of the state $Have(cake)$.

The graphplan algorithm is based on [29] and is illustrated in algorithm 2. It uses the planning graph not to find a heuristic, but rather to extract a plan directly. It starts by taking the planning problem as an input, and then repeatedly adds a supplementary level by calling EXPAND-GRAPH to the problem's planning graph until all of the goal states are represented non-mutex. When all goals are represented non-mutex the algorithm calls EXTRACT-SOLUTION in order to find a plan which solves the problem. If it is not solvable, the algorithm expands another level with the use of EXPAND-GRAPH. It continues

repeatedly until the termination condition is satisfied.

Algorithm 2 Graphplan

```
1: function GRAPHPLAN(problem)
2:     graph ← INITIAL-PLANNING-GRAPH(problem)
3:     goals ← CONJUNCTS(problem.GOAL)
4:     nogoods ← an empty hash table
5:     for tl = 0 to ∞ do
6:         if goals all non-mutex in S_t of graph then
7:             solution ← EXTRACT-SOLUTION(graph, goals, NUM-
   LEVEL(graph), nogoods)
8:             if solution ≠ failure then
9:                 return solution
10:        if graph and nogoods have both leveled off then
11:            return failure
12:        graph ← EXPAND-GRAPH(graph, problem)
```

A simple example to which can be formulated using a planning graph is the ”have-cake-eat-cake” problem. The planning problem with its domain is described in listing 2.2, with figure 2.3 illustrating the planning graph itself. The domain is described using the de facto standard in modeling planning problems for AI planning methods, i.e. Planning Domain Definition Language (PDDL). More details on the modeling language is to be found in chapter 3.

Figure 2.3 illustrates the alternating levels of actions, A_i, and states, S_i. S_0 denotes the initial state as also defined in listing 2.2. The rectangles represent all the possible actions, while the small squares describe persistent actions which essentially does not change the current state. Furthermore, the grey curved lines are examples of mutex relations either between states or actions.

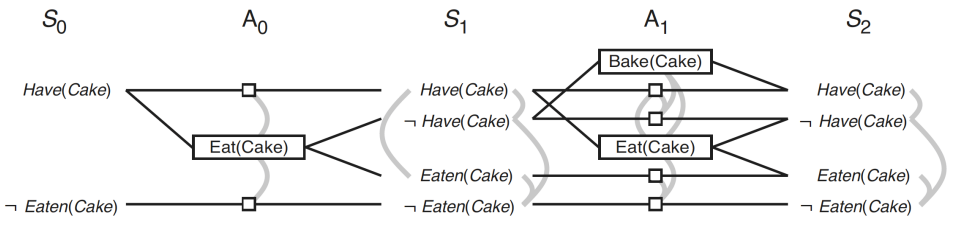


Figure 2.3: Example of a planning problem represented as a planning graph [29]

```
(define (problem have-cake-eat-cake)
    (:domain cake-world)
    (:types cake)
    (:predicates
        (have ?c - cake)
        (eaten ?c - cake)
    )
    (:init
        have(cake) and
        not eaten
    )
    (:goal
        have(cake) and
        eaten(cake)
    )
    (:action eat
        :parameters (?c - cake)
        :precondition (have(?c))
        :effect (not have(?c) and eaten(?c))
    )
    (:action bake
        :parameters (?c - cake)
        :precondition (not have(?c))
        :effect (have(?c))
    )
)
```

Listing 2.2: Description of planning problem in the modeling language PDDL

2.1.2 Plan-Space Planning

A plan space is another alternative for modelling a planning problem to the earlier introduced state space. Plan-space planning constitutes a more elaborate search space, especially compared to state-space planning, where the search space is directly given by the state-transition system Σ. The nodes in the plan space are defined as partially specified plans, while the connecting arcs are defined as plan refinement operations. A partial plan, also called partial-order plan, $\pi = (A, \prec, B, L)$ is defined as a solution plan for a planning problem $P = (\Sigma, s_0, g)$. In terms of variables used to describe the planning problem; as earlier, Σ represents the state-transition system while s_0 and g represent the initial and goal state, respectively. For the partial plan, π, A is the set of ordered actions, while $\prec$ represents the ordering constraints for these actions. Finally, B represents the binding constraints for the partial plan and L is the set of causal links [10].

The earlier mentioned plan refinements operations includes operations which contribute in completing a partial plan by removing possible inconsistencies or to achieve an open goal, i.e. a reachable goal state. The plan refinements operations follow the least commitment

principles, which means that only strictly needed constraints are added to the partial plan [10]. This means that unnecessary constraints are not added to the plan, hence avoiding over-complications.

Plan-space planning can also be distinguished from the earlier introduced state-space planing by the definition of its solution plan. Plan-space planning has more generalized plan structures compared to state-space planning [10]. By using constraint-satisfaction techniques, i.e. techniques which ensure that the defined constraints are satisfied, the resulting solution of the plan space search is more flexible compared to linear sequences of ground actions which is the case for solutions of state space search problems [27]. The solution of a plan-space planning problem results in a more generalized plan, since it considers planning as two operations, i.e. the choice of actions and ordering the already chosen actions to achieve the pre-defined goal. Hence, a plan does not necessarily result in a sequence of actions, but rather a set of planning operators with both ordering and binding constraints [10].

2.1.2.1 Hierarchical Task Planner

Hierarchical Task Network (HTN) is a planning method which is quite similar to classical planning in the way it is structured. This includes that the states in the world can be described by a set of atoms, i.e. variables with only positive measures, and actions as corresponding state-transitions. However, HTN can be differentiated from classical planners, which have been explored earlier, by what the method plans for and how it plans for it [10].

Another distinction between classical planners and HTN, is that HTN's objectives are based on executing a set of tasks opposed to achieving a goal state. A task is defined as something which needs to be done, thus a task network is a set of tasks. Tasks can be divided into two subcategories: primitive and non-primitive tasks. The first defines tasks which can be performed directly, while the latter describes tasks where the planner needs to find out and determine how to perform them [33]. An HTN planner divides the non-primitive tasks into sub-tasks in order to determine how to execute the non-primitive tasks. An HTN planner's objective is also to produce a sequence of action which will result in the execution of a task. This resembles the classical planners from earlier, but it is important to distinguish between reaching a goal state and being able to achieve a task [34].

An HTN planner also requires a set of operators and methods to solve a planning problem. Operators describe the effects of each task, while the methods describe how the planner could perform various non-primitive tasks. Each method can be defined as a pairing between a task t and task network d, resulting in the method $m = (t, d)$. The task network defines the sub-tasks which are necessary to perform in order to achieve the main task t. The only condition for the task network d is that all the sub-tasks in the set must satisfy the constraints enforced upon the system [33].

A classical example of a planning problem that is the "travel" example, described as a hierarchical task network and then solved as illustrated in figure 2.4. Table 2.1 gives an

overview of the different states that are reached when solving the planning problem. From one state to the other, the changes are marked in bold. The main task is denoted as the initial task in figure 2.4 and represents the overall task one wants to solve in this example, which is for ”me” to travel from my home to the park. There are two possible methods for achieving the initial task; ”travel-by-foot” and ”travel-by-taxi”. These are categorized as methods, since they are considered non-primitive tasks. The methods are therefore further divided into sub-tasks, i.e. ”call-taxi”, ”ride” and ”pay-driver”. These sub-tasks are at the lowest level and are therefore considered operators. Finally, there are ordering constraint on the mentioned operators which enforces them to be executed in a certain order, i.e. ”call-taxi”, then ”ride” and finally ”pay-driver”.

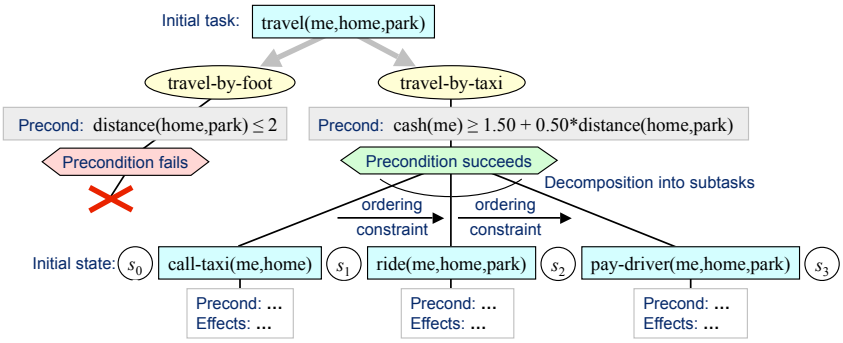


Figure 2.4: Example of a problem space for HTN [29]

$\mathbf{s_0}$	location(me)=home, cash(me)=20, distance(home, park)=8
$\mathbf{s_1}$	location(me)=home, **location(taxi)=home**, cash(me)=20, distance(home, park)=8
$\mathbf{s_2}$	location(me)=**park**, location(taxi)=**park**, cash(me)=20, distance(home, park)=8
$\mathbf{s_3}$	location(me)=park, location(taxi)=park, cash(me)=**14.50**, distance(home, park)=8

Table 2.1: Overview of the different states in the HTN planning problem

2.1.2.2 Simple Hierarchical Ordered Planner

Simple Hierarchical Ordered Planner (SHOP) is a planner based the HTN planner principles. The main difference which distinguishes SHOP from HTN is the fact that SHOP uses ordered task decomposition as control strategy. This strategy breaks the main tasks into smaller subtasks and then generates the plan's actions in the exact same order as they will be executed by the system [35]. By having all the actions ordered, a big part of the existing uncertainty about the world model is removed [34].

The second version of Simple Hierarchical Ordered Planner is simply called SHOP2, and is a further development of SHOP. One of the main drawbacks of SHOP is that the method requires the tasks to be totally ordered, which to some extent is considered restrictive. This requirement prevents the method from potentially interleaving sub-tasks which might be excessive in order to solve the overall task. Hence, SHOP2 accepts the use of tasks which are only partially ordered. Consequently, SHOP2 can potentially solve planning problem faster than SHOP as it can interleave sub-tasks which are not strictly necessary [36].

2.2 Machine Learning

Machine Learning (ML) is defined as an application of Artificial Intelligence (AI) in which a system in a changing environment is able to learn, adapt and consequently improve its performance without being programmed explicitly [37]. A system is able to learn through a software agent, whose main purpose is to act and consequently learn on the system's behalf [38]. The software agent plays the same role as the aforementioned artificial agent in AI Planning. Machine Learning can in general be categorized into three main approaches; supervised learning, unsupervised learning and reinforcement learning.

Supervised learning is that the software agent is given some examples of corresponding input and output, and thereby learns how to map the input to the output by finding such a function. Examples of supervised learning are for instance classification and regression, as illustrated in figure 2.5. The latter is mainly used for output represented by continuous values while classification is used for categorical output. Unsupervised learning is when the software agent is given some input, in which it is able to find some patterns, regardless of receiving any explicit feedback. This is often done through clustering, which detects groups of data which are similar to each other while also detecting outliers, i.e. data which deviates from the clusters [29]. An example of several clusters and remaining outliers is depicted in figure 2.6.

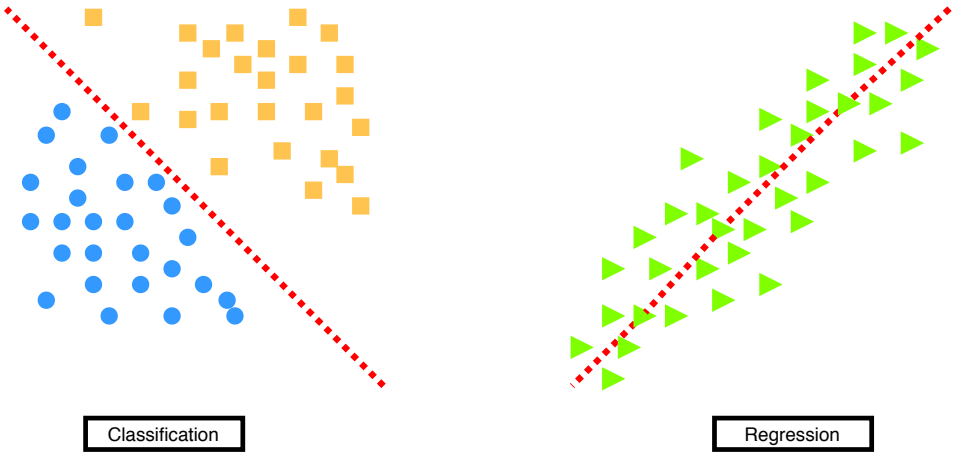


Figure 2.5: Examples of supervised learning [39]

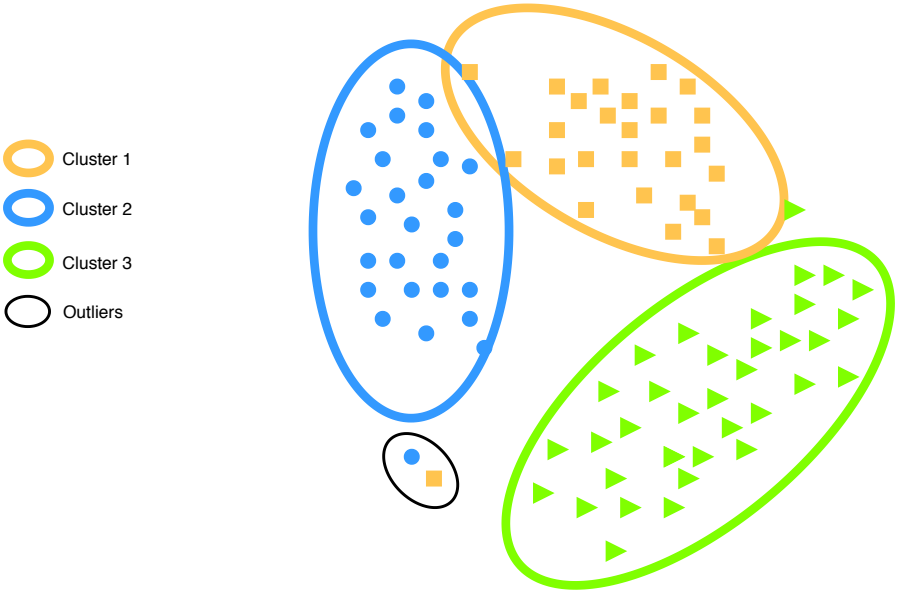


Figure 2.6: Example of unsupervised learning [39]

Another group of machine learning algorithms worth mentioning is semi-supervised learning. Essentially, this is considered a merge between supervised and unsupervised learning. The algorithms in semi-supervised learning are given both known and unknown data, as for supervised and unsupervised learning, respectively. An advantage of combining both known and unknown data, is the potential increase in learning accuracy [39].

The last category, i.e. Reinforcement Learning (RL), is the aspect of machine learning which is used in the works of this thesis and will be the main focus among the machine learning categories. It is further explained in section 2.2.2. Before going into the definition of Reinforcement Learning and its use cases, some basic notions of Markov Decision Processes are necessary.

2.2.1 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework for modelling sequential decision-making processes. A sequential decision-making process is essentially a sequence of problems in which the software agent needs to decide which actions to take by also evaluating how it would affect the subsequent problems [40]. The main purpose of MDPs are to model a problem where the agent learns through interaction with the environment in order to achieve a goal [41].

Markov Decision Processes can be described as the following 4-tuple: (S, A, p, r), where S represents the state space of the process and includes all the possible states that can be reached by the agent. The possible actions are represented by the set, A, in the aforementioned tuple. These control the dynamics of the states. Furthermore, p represents the state transition function and can also be formulated in the following manner: $P(s'|s, a)$. The state transition function describes the probability of ending up in some state s' given an action a taken from a state s [29]. Lastly, r describes the resulting reward based on the state transitions. This is the reward used by the agent to determine if the taken action is beneficial towards achieving the desired goal [40].

The solution of a Markov Decision Process is called a policy, and it describes what actions the software agent should take in any given state. Such a policy is often denoted π, while $\pi(s)$ denotes the recommended action corresponding to the solution in the state s. Moreover, an optimal solution to an MDP is the policy which ensures the highest amount of expected utility. This policy is denoted π^* and is called the optimal policy. As for a general policy, the recommended action in a given state s is denoted in a similar manner; $\pi^*(s)$ [29].

A simple example of a problem formulated as a Markov Decision Process is illustrated in figure 2.7. It describes a person's state of mind tomorrow, given the same person's state of mind today. The earlier introduced 4-tuple for MDP's can be used to describe the example, i.e. (S, A, p, r). The states S are illustrated as circles in 2.7, while the actions $a_i \in A$ are illustrated as arrows. Furthermore, the numbers marked along the actions are the state transition probabilities p. Table 2.2 sums up the these state transition probabilities where the first column represents the starting state and the columns represent the end state. For instance the state transition probability from the state of mind "bored" one day to the state of mind "tired" the following day is 0.5. Finally, it is worth mentioning that in the actual example there are no explicit rewards r.

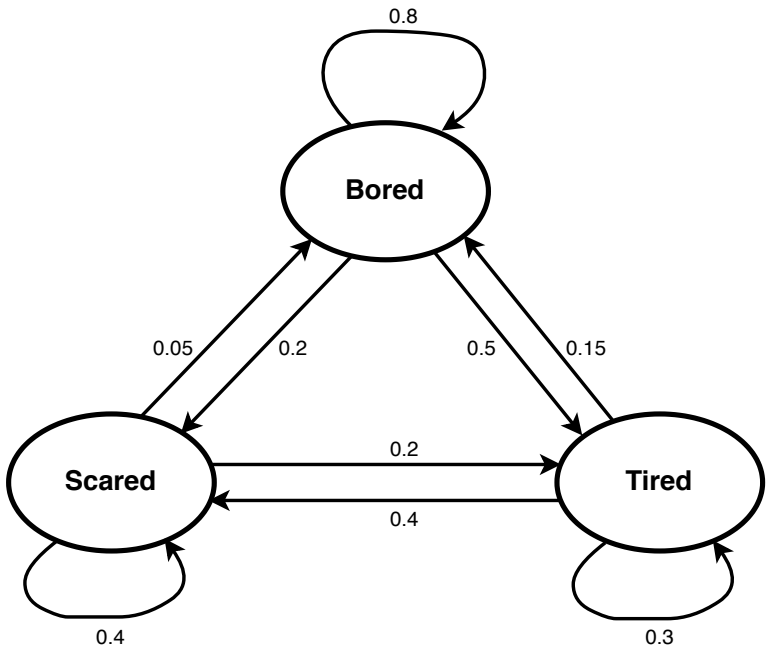


Figure 2.7: Example of a problem modelled as a Markov Decision Process [38]

	Bored	**Scared**	**Tired**
Bored	0.8	0.2	0.5
Scared	0.05	0.4	0.2
Tired	0.15	0.4	0.3

Table 2.2: The state transition probabilities for the example in figure 2.7

2.2.2 Reinforcement Learning

Reinforcement Learning is, as mentioned previously, another important approach within Machine Learning. This approach can be distinguished from the aforementioned supervised and unsupervised learning, as the software agent learns by the feedback gained by some reinforcements. Such reinforcements can be modeled as both punishments and rewards, which reflects if the software agent is acting in a beneficial manner or not. The aim in reinforcement learning is to maximize the reward while achieving the desired goal [29].

Figure 2.8 illustrates the interaction between the software agent and the environment. The sub-scripted t in the figure denotes the time steps at which the agent and environment interacts with each other. This interaction consists of the agent receiving some representation of the current state $s \in S$ of the environment. Based on this state $s \in S$, an action $a \in A$ is selected, which results in the agent receiving the reward R_{t+1}. Thereby, it also reaches

a new state which is denoted S_{t+1} in the figure [41].

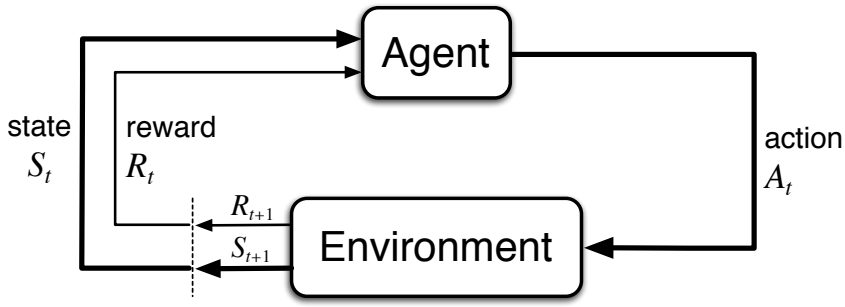


Figure 2.8: Interaction between software agent and environment in a Markov Decision Process [41]

Generally, reinforcement learning is used for solving problems which are modeled as the previously introduced Markov Decision Process (section 2.2.1). Thus, the main difference in the modelling of an MDP problem and an RL problem is that for the latter neither the reward function r nor the state transition probability function p are known a prior to execution [40].

Like for MDPs, the solution of a RL problem is also called a policy and is found by maximizing the expected total reward [29]. The expected total reward is also known as the value and can be computed as in equation (2.1) or (2.2). $V(s)$ in equation (2.1) represents the value function as a state-value function, which means that the value function uses the current state s_t of the agent and the average over all the possible actions in that state. Furthermore, equation (2.2) uses the current state s_t and also the each of the possible actions a_t separately. This value function is also called an action-value function [38]. Both the action-value and state-value functions are dependent of a constant γ which denotes the discount rate between $0 \leq \gamma \leq 1$. The discount rate represents the importance of current steps compared to future steps. For instance, for $\gamma = 0$ the future rewards are omitted and only the immediate rewards are considered [37].

$$V(s) = E(r_t|s_t = s) = E\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}\ s_t = s\right\} \tag{2.1}$$

$$Q(s,a) = E(r_t|s_t = s, a_t = a) = E\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}\ s_t = s, a_t = a\right\} \tag{2.2}$$

As for MDPs, the solution to a RL problem is called a policy and denoted π. In a similar manner, the optimal policy is denoted π^* and can be found by choosing the policy which maximizes the value function [37]. This is described in equation (2.3).

$$V^* = max_\pi V^\pi(s_t) \qquad \forall s_t \tag{2.3}$$

Equation (2.4) shows the Bellman Equation which is used in RL to find the action which maximizes the value function. The second part of the equation, i.e. $P(s_{t+1}|s_t, a_t)$ describes the probability of moving from the current state s_t to the next state s_{t+1} by taking action a_t. Afterwards the agent follows the optimal policy and $V^*(s_{t+1})$ represents the cumulative reward which is expected. Furthermore, all the possible states that can be reached are summed together and discounted by γ. Finally, the immediate reward, $E[r_{t+1}]$ is added to result in the total expected cumulative reward for the action a_t [37].

$$V^*(s_t) = max_{a_t} \left(E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) V^*(s_{t+1}) \right) \tag{2.4}$$

One can distinguish between model-free and model-based reinforcement learning methods, based on if the methods build a model of the functions describing both the reward and state transition. A model-free method does not model the the reward and state transition functions and are therefore dependent on the agent exploring the environment. The value function for model-free methods are thereby updated locally as the agent explores. A model-based method on the contrary have known reward and state transition functions and consequently the model of the problem is complete. In this case the value function can be computed as in the aforementioned equations, i.e. equations (2.1) and (2.2) [40].

Another central concept within Reinforcement Learning is Temporal-Difference (TD). TD methods are able to learn regardless of having a model of the dynamics of the environment, which make the methods model-free [41]. Furthermore, TD methods update the current state value by using the discounted value, where the discount is γ, of the next state and reward. This is done by the method by comparing the estimated value of a state and the obtained discounted value and thereby updating the value of the state. Therefore, TD methods only need until reaching the next step before updating the value for a state. Such methods are also called bootstrapping methods as they update based on a known estimate, as well as doing it in steps [37].

Equation (2.5) describes how a simple TD method updates its value. As mentioned previously, a TD method updates its value by reaching the next step. This means that at time-step $t + 1$, the method can update its value for the current state by using the value for r_{t+1} which is observed and the estimated value for the next state $V(s_{t+1})$. [41].

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.5}$$

Within reinforcement learning methods there is generally a trade-off between exploration and exploitation, which is caused by the dilemma if the agent should explore further or act on the already discovered information [42]. Through exploration the agent is able to get information which benefits its long-term well-being. Having an agent which only prioritizes exploration would result in having a lot of knowledge about the environment, but not putting it to use. On the other hand, an agent who prioritizes exploitation would act based on whichever action is beneficial in the moment without taking into account later possibilities. This could also get the agent stuck [29]. An example of a method which makes a trade-off between exploration and exploitation to decide which action to choose is the ϵ-greedy method. Moreover, it is used in the Q-learning algorithm and is described below.

One can also distinguish between on-policy and off-policy learning algorithms. An on-policy learning algorithm updates the values and also estimates the total reward based on the fact that the chosen policy is followed from start to end. An example of such an algorithm is called State-Action-Reward-State-Action (SARSA) On the contrary, an off-policy learning algorithm does not necessarily assume that the current chosen policy will be followed, and selects actions based on whichever maximizes the values [37]. An example of an off-policy algorithm is Q-learning, which is further explored in the upcoming section.

2.2.2.1 Q-learning

Q-learning is an off-policy TD learning algorithm, making it a model-free algorithm. The main idea behind the Q-learning is that the algorithm uses the action-value function, $Q(s, a)$ to approximate the optimal action-value function q_*. The resulting value from the action-value function is also called the Q-value. Algorithm 3 gives an overview of how Q-learning works [41].

As a starting point the Q-value, i.e. $Q(s, a)$, for the initial state s is initialized arbitrarily, while the Q-value for the terminal state $s_{terminal}$ is set to 0. As this is the terminal state, the Q-value is no longer dependent on any action a. Furthermore, the algorithm loops through the predefined number of episodes. The number of episodes represents the number of sequences the algorithm, and thereby the agent, goes through in order to learn about the environment. The episode is terminated when the termination condition is reached, for instance upon reaching the desired objective.

For each of the episodes the state variable s is initialized, and then the algorithm performs one step at a time. A step starts by verifying if the current state corresponds to the terminal state/objective. If this is the case, the current episode is terminated. If not, an action a is chosen based on some arbitrary policy. An example for such a policy is the ϵ-greedy policy, which essentially chooses a random action a based on the probability defined by the variable ϵ. This corresponds to the exploration aspect of the algorithm. For the probability defined by $1 - \epsilon$, the algorithm chooses the action that seems the most reasonable based on the obtained information up until that point. This represents the exploitation aspect of the ϵ-greedy method.

Based on the action chosen by the ϵ-greedy method, the agent is able to observe the next state and the reward. Thereby, it uses all the obtained information to update the Q-value using the previously defined Q-value, the learning rate α, the reward r, the discount rate γ and the action which maximizes the future expected reward given the next state s'. Finally, the current state s is updated [41].

Algorithm 3 Q-learning

Initialize $Q(s, a)$ arbitrarily and $Q(s_{terminal}, \cdot)$ = 0
for each episode **do**
    $s \leftarrow$ initial state
    for each step of episode **do**
        while s $\neq s_{terminal}$ **do**
            $a \leftarrow$action from policy
            Take action a, observe r, s'
            $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_a Q(s', a) - Q(s, a)]$
            $s \leftarrow s'$

Chapter 3

Tools and Libraries

This chapter will give an overview of the modeling tools and libraries that have been used for solving the planning problems which will further on be introduced in Chapter 4. Additionally, this chapter gives an overview of how the different libraries for the different methods are structured with respect to different files and implementation of the mission planning problem.

3.1 Modeling Language

A modeling language is an essential tool for gathering known information about a system or environment in a structured manner, including rules, constraints or description of valid actions [43]. For instance, how an agent or actor would behave in an environment could be described using a modeling language, in addition to the environment itself. The upcoming section describes a modeling language called Planning Domain Definition Language which is used for describing the planning problem for AI Planning methods.

3.1.1 PDDL

Planning Domain Definition Language (PDDL) is a language used to describe the physics of a planning domain. This includes possible predicates within the domain, the possible actions, the structure of compound actions and what the resulting effects are of each action. Furthermore, many planners also require some additional information about which actions to carry out in order to achieve some goal, which can be interpreted as an advice to the planner. The PDDL syntax omits this additional information to retain some neutrality. Every planner compensates for this lack of information by extending the notation of PDDL to fit its own use cases. Hence, there will be great variations in how different planners would extend the use of PDDL notation [44].

PDDL can be used to describe a planning problem, where the state of a world can be described based on a collection of variables [44]. As mentioned earlier, a search space requires four attributes; an initial state, the available actions in a certain state, the effects of performing an action and the goal state [29].

A simple example of how PDDL can be used to describe a planning problem is described in listing 3.1 and listing 3.2. This planning problem is described by a domain, the objects within the domain and their type. The listing also include a list of predicates, which represent the properties of the defined objects. Furthermore, the listing also defines an initial and a goal state which describe which state the domain starts in and the desired state, respectively. Finally, the planning problem defines possible actions which are used for going from the initial state and reaching the desired goal state.

The corresponding illustration of the aforementioned example is shown in figure 3.1, with both the initial state and the desired goal state.

```
(define (problem dock-worker-robot )
    (:domain dock-worker-robot-world)
    (:types
        robot
        docks
        container
        constant
    )
    (:objects
        r1 - robot
        d1, d2, d3 - docks
        c1 - container
        nil - constant
    )
    (:predicates
        (cargo ?r - robot)
        (loc ?c - container)
        (loc ?r - robot)
    )
    (:init
        loc(r1) = d3 and
        cargo(r1) = nil and
        loc(c1) = d1
    )
    (:goal
        loc(r1) = d3 and
        loc(c1) = r1
    )
)
```

Listing 3.1: Domain description of the dock-worker-robot example in PDDL

```
(:action load
    :parameters (?r - robot
                 ?c - container
                 ?l - location)
    :precondition(cargo(r) = nil and
                  loc(c) = l and
                  loc(r) = l)
    :effect(cargo(r) = c and
            loc(c) = r
)
(:action unload
    :parameters (?r - robot
                 ?c - container
                 ?l - location)
    :precondition(cargo(r) = c
                  loc(r) = l)
    :effect(cargo(r) = nil
            loc(c) = l
)
(:action move
    :parameters(?r - robot
                ?d - dock
                ?e - dock)
    :precondition(loc(r) = d)
    :effect(loc(r) = e
)
```

Listing 3.2: Action description of the dock-worker-robot example in PDDL

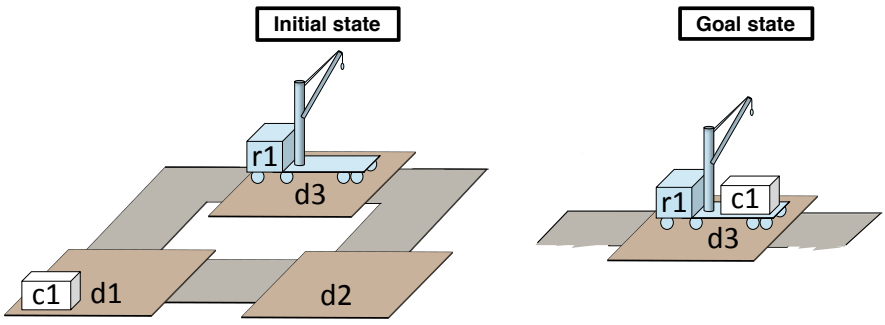


Figure 3.1: The corresponding world setup with both initial and goal state of the PDDL described planning problem [27]

3.2 Libraries

All the used libraries are based on the programming language Python. The main reason behind this choice is the potential long-term aspect of this project, where one could possibly use a simulator to solve the mission planning problem. Numerous existing, open-source simulators for Unmanned Underwater Vehicles (UUV) are compatible with programs written in Python, which would ease the transition from the current Python program to using it in a simulated environment. Furthermore, the choice of one single programming language for all the libraries is related to consistency within the works of this project. The reader only needs to master one programming language when potentially replicating the results or developing them further.

This chapter will go into further details of how the libraries are structured, in addition to how they are used in the scope of this thesis. The remaining of this chapter describes each of the libraries for each of the implemented methods. The actual implementation and the details regarded to it are to be found in the following chapter, i.e. chapter 4.

3.2.1 Graphplan

The Graphplan algorithm is implemented based on the previously mentioned algorithm 2 in section 2.1.1.3, for which there exists an online resource at GitHub [4]. The GitHub-repository, ”aima-python”, contains a Python-implementation of the majority of the algorithms introduced in [29], including the Graphplan algorithm.

In order to implement the Graphplan algorithm for a planning problem using the aforementioned library, eight files from the overall library ”aima-python” are extracted. The following files are necessary from ”aima-python”: **agents.py, csp.py, logic.py, planning.py, probability.py, search.py, test.py** and **utils.py**. The remaining files in the library are not required for using the Graphplan algorithm and are therefore omitted in the works of this thesis.

The main code for the Graphplan algorithm is implemented in **planning.py**. The most important classes in this file are `PlanningProblem` and `GraphPlan`. The first class is used for constructing a planning problem, for instance as defined by the PDDL representation of the domain. In order to define a planning problem an initial state, a goal state and possible actions need to be defined. This is also the case for the implementation of the self-defined planning problem, which will be introduced and further discussed in chapter 4. Thereafter, the planning problem is redefined as a graph by the `GraphPlan` class so that it may be solved using the Graphplan algorithm. Following the creation of the graph with the associated states and actions, the `Graphplan` class is able to return a solution to the planning problem.

The file **test.py** contains the code for testing the Graphplan algorithm. This is done by defining an initial state and a desired goal state for the planning problem. Thenceforth, the planning problem is created and solved. Finally, the solution is linearized in order to return a solution which is readable for the user and clearly states the different actions necessary

for going from the initial state and to reach the predefined goal state.

The remaining of the aforementioned files from the ”aima-python” library contains helper-functions used by the main `Graphplan` class in order to solve a planning problem based on the Graphplan algorithm. Among others, these helper-functions include the ability of creating suitable structures for both states and actions, determine which actions are executable based on the satisfied preconditions and some logic for distinguishing literals from negated literals.

The ”aima-python” library was chosen due to several reasons. Firstly, it met the requirements related to the programming language. Secondly, it was based on a resource which was heavily used in the research of the Graphplan algorithm. The implementation was also liable, as it was implemented by the authors of the aforementioned resource [29].

3.2.2 Hierarchical Task Network

The used library for implementing the plan-space planner Hierarchical Task Network(HTN) is based on a Simple Hierarchical Ordered Planner (SHOP). As mentioned previously in section 2.1.2.2, the Simple Hierarchical Ordered Planner can be distinguished from classical Hierarchical Task Network by providing an ordered solution. This means that the ultimate solution consists of actions to be executed in the correct order, i.e. from start to end. The used library for the SHOP method is developed in Python by Dana S. Nau and retrieved from BitBucket [3].

Altogether, the library contains six files. However, only one of them is necessary for implementing the HTN planner. The remaining are classical examples from AI planning, such as the blocks world problem as introduced in section 2.1.1.1 for the STRIPS method and the simple travel problem as introduced for the HTN method in section 2.1.2.1. The first example is already illustrated in figure 2.1 and the latter in figure 2.4. The respective Python files in the SHOP library for the block world problem are as following: **blocks_world_examples.py**, **blocks_world_methods.py**, **blocks_world_methods2.py**, **blocks_world_opera** while for the simple travel example: **simple_travel_example.py**.

The essential file when implementing the HTN method using this SHOP library is **pyhop.py**. This file contains all the necessary helper-functions and classes for implementing the planning problem as tasks and thereafter solving it. The main file, **pyhop.py**, contains two separate classes `State` and `Goal` which are used to define the initial state and goal state, respectively. This is in a similar manner as for the Graphplan algorithm introduced in the previous section.

Furthermore, **pyhop.py** also contains helper-functions for defining operators and methods, which are essentially the primitive tasks and non-primitive tasks, respectively. For the sake of neatness, the methods and operators will be implemented in separate files. The structure of the implemented files is explained thoroughly in section 4.5.1 in the upcoming chapter.

Finally, the most important function in **pyhop.py** is `seek_plan()` as this command tries to solve the planning problem based on the defined state, tasks and plan. The latter corresponds to the current partial plan, which is a plan for solving for instance one single defined task. The sum of all partial plans constitute the ultimate solution.

There were several other reasons for choosing this library, in addition to the fact that the library was written in Python. Firstly, it was well-documented and had good readability. Secondly, the library was created by Dana Nau, who is considered as one of the pioneers in the field of AI planning and especially within state-space planning.

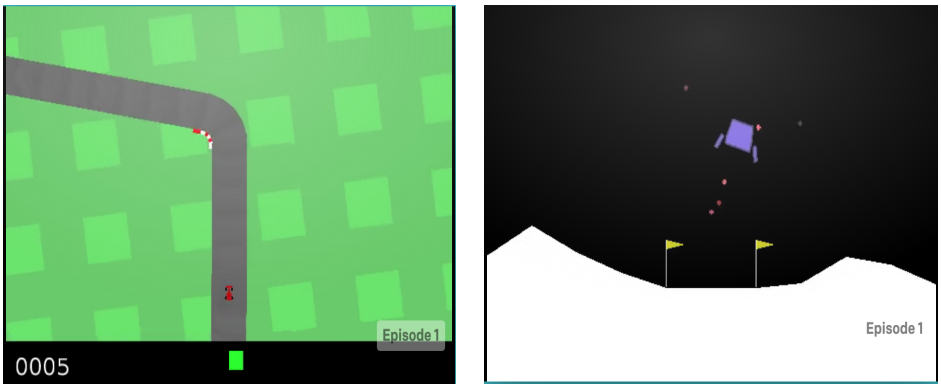
3.2.3 Q-learning

The only machine learning algorithm used in the works of this thesis is Q-learning. It differs from the other aforementioned methods and their respective libraries, as there is no existing library that can directly be used for implementing the Q-learning algorithm. There are several resources available for the algorithm itself, but they are not compatible with the environment which had to be created. More details on the environment can be found in section 4.5.3. Hence, this section will give an overview of which libraries were used in order to create the required reinforcement learning environment, as this is a prerequisite for the algorithm itself.

The reinforcement learning environment is based on [45] and retrieved from the corresponding online resource on GitHub [5]. The custom environment is created in a similar manner as the pre-existing environments which are a part of the OpenAI Gym. The OpenAI Gym is a toolkit which is mainly used for reinforcement learning [45]. By default, the toolkit contains several environments which already have a common interface. By creating an environment which is compatible with the OpenAI Gym, already implemented tools from the toolkit would be more easily compatible.

In order to create a custom environment, which may be both partially or fully observable based on how much information the software agent has access to, there are a few Application Programming Interface (API) methods which have to be included. These are: `step()`, `reset()`, `render()`, `close()` and `seed()`, and it is recommended that all custom environment also contains these API methods [45]. The method `step()` runs one step, i.e. time step, in the dynamics of the defined environment. Secondly, `reset()` resets the entire environment and sets the current state back to the initial state. Following, the method `render()` provides feedback regarding the environment to the user. The API method `close()` is optional, but gives the environment the option to perform any necessary cleanup. Finally, the `seed()` method is used for setting the seed for the random number generator which would be used by the custom environment.

The library [5] was first and foremost chosen based on the available documentation related to creating a custom environment, which was the case for the works of this thesis. Another consideration in the choice of library was the long-term aspect of the project. OpenAI Gym is under continuous development and have potential for implementing an appropriate simulator with Graphic User Interface (GUI). Examples of existing GUI's for the OpenAI Gym are illustrated in figure 3.2.



(a) OpenAI Gym Car Environment [45] **(b)** OpenAI Gym Lunar Environment [45]

Figure 3.2: Examples of existing Graphic User Interfaces for the OpenAI Gym

Chapter 4

Industrial Subsea Mission Definition and Implementation

This chapter will introduce the subsea mission which is defined with the industrial feedback obtained from the ROV operators Jon Englund and Peter Baastad at Oceaneering. It is worth mentioning that the defined mission is an attempt at reflecting a real mission, but is obviously simplified in the scope of this thesis. Furthermore, this chapter will describe the different aspects of the mission which will be solved by the previously introduced methods, i.e. Graphplan, HTN and Q-learning. This chapter will also explain which design-related decisions are made for a simpler implementation. Finally, implementation of the mission planning problem using the previously introduced libraries and tools is described.

4.1 Industrial Subsea Mission Definition

As a starting point, a domain for the subsea mission in the context of Inspection, Maintenance and Repair (IMR) operations was defined. The environment for the defined mission are a few subsea installations on the seabed, which include a docking station for the Underwater Intervention Drone (UID), a panel with several valves, a pipeline and a warehouse. An overall illustration of the setup is shown in figure 4.1. The different arrows illustrates the potential path of motion of the UID.

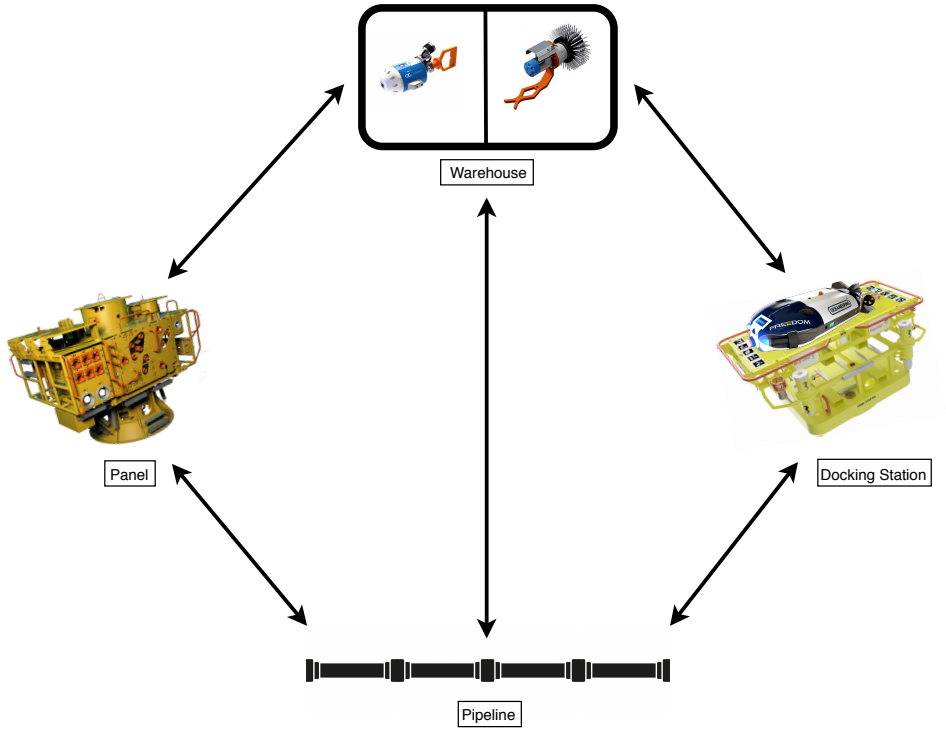


Figure 4.1: Overall structure of the subsea installation used in the planning problem [2]

The docking station is considered a garage for the UID, so that it is able to charge its batteries and be safely stored whenever it is idle, i.e. not performing any operations or missions. The UID docks to the station upon completion of mission so that it is securely fastened to an installation and not able to move unless undocked. This prevents the UID from moving unnecessarily and potentially colliding due to unexpected disturbances caused by wave and current forces. Additionally, the docking station is the location where the UID is able to communicate with the onshore operation room from where the UID is continuously monitored.

Another part of the overall subsea installation is the panel, which consists of numerous valves which could for instance control the flow of different substances. The valves can be operated by the UID assuming that the correct tool is installed on it. Furthermore, the panel also has built-in bars so that the UID can dock to the panel and be stable when operating or inspecting the valves.

As illustrated in figure 4.1 the overall subsea installation also consists of a pipeline. The function of such a pipeline could be to transport oil or gas to an onshore installation, for instance an oil refinery [46]. Normally, the pipelines span across several kilometres, but in this current environment only a section of the pipeline is included.

The warehouse is an additional location at the subsea installation where the UID can install or change tools based on the requirements of the mission or operation. An example of such a tool is the manipulator torque multiplier tool, whose main function is to prevent damages on the valves which are caused by over torque [47]. Another commonly used tool in IMR operations is the cleaning brush, which is used to clean the surface of an installation. An example of the use of it might be when UID is trying to inspect or operate a valve, but the valve is covered by marine growth [48]. The UID then cleans the valve and subsequently operates it.

(a) Multipurpose cleaning tool with round brush [48] **(b)** Manipulator torque multiplier tool [47]

Figure 4.2: Examples of tools for the UID

Based on the previously introduced environment with the installations, one may formulate an IMR mission. The IMR mission can either be an inspection mission or an intervention mission. It is also noteworthy that the overall mission could be a combination of both inspection and intervention, but in that case the mission is divided accordingly. Hence, the inspection part could be solved first and followed by the intervention part.

The upcoming sections will go into details about how a mission is defined and afterwards formulate it as a problem which can be solved with the previously introduced methods. The problem is formulated based upon input gained by the industry, i.e. the ROV operators Jon Englund and Peter Baastad at Oceaneering.

4.2 Formulation of the Mission Planning Problem

The mission planning problem is defined within the domain described in the previous section and illustrated in figure 4.1, with some minor modifications. As a starting point the pipeline is omitted altogether from the created scenario. Hence, the environment only contains the docking station, the warehouse, the panel with multiple valves and the UID. The modified setup is illustrated in figure 4.3.

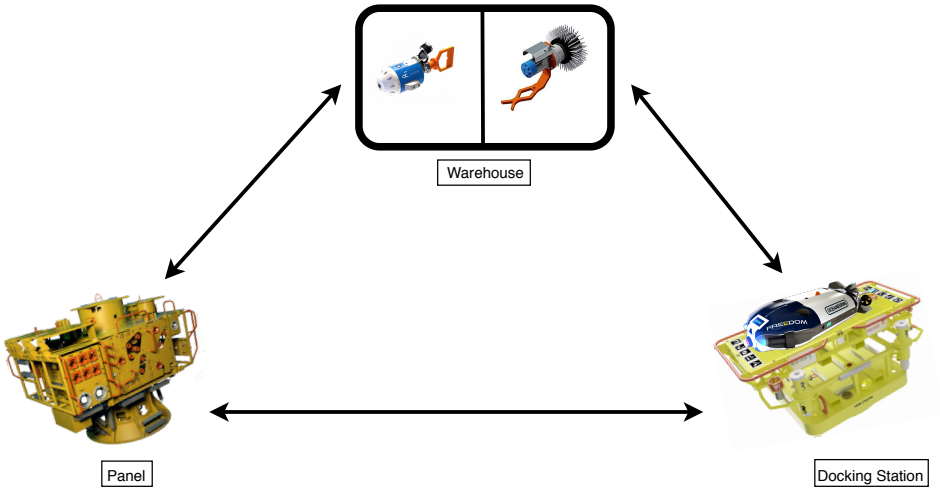


Figure 4.3: Overall structure of the subsea installation used in the planning problem

In the first scenario the mission starts with the UID being located at the docking station. It is then commanded by the operation center to operate valve 1 on the panel. It is worth mentioning that in general the operation center can be both onshore and offshore. In order to operate the desired valve, the UID has to move from the docking station to the panel. The operation center has communicated to the UID and clarified that valve 1 needs be operated. The UID docks to the panel for stability, and prepares for operating the valve. Before arriving at the panel and observing the valve, the UID has no information regarding which manipulator tool is necessary for the operation. Hence, it needs to check if it has the correct tool installed for operating the required valve upon docking. If the correct tool is indeed installed, the UID can proceed with the operation. If the correct tool is not installed, the UID needs to move to the warehouse, change tool, return back to the panel and dock to it. It is then ready for operating the valve of interest. Upon completion of operating valve 1, the UID must inspect the remaining valves and update their statuses. Finally, the UID can move back and dock to the docking station until the next command from the operation center is received.

The overall structure of the introduced mission is illustrated in figure 4.4 as a flowchart with the corresponding actions necessary for solving it.

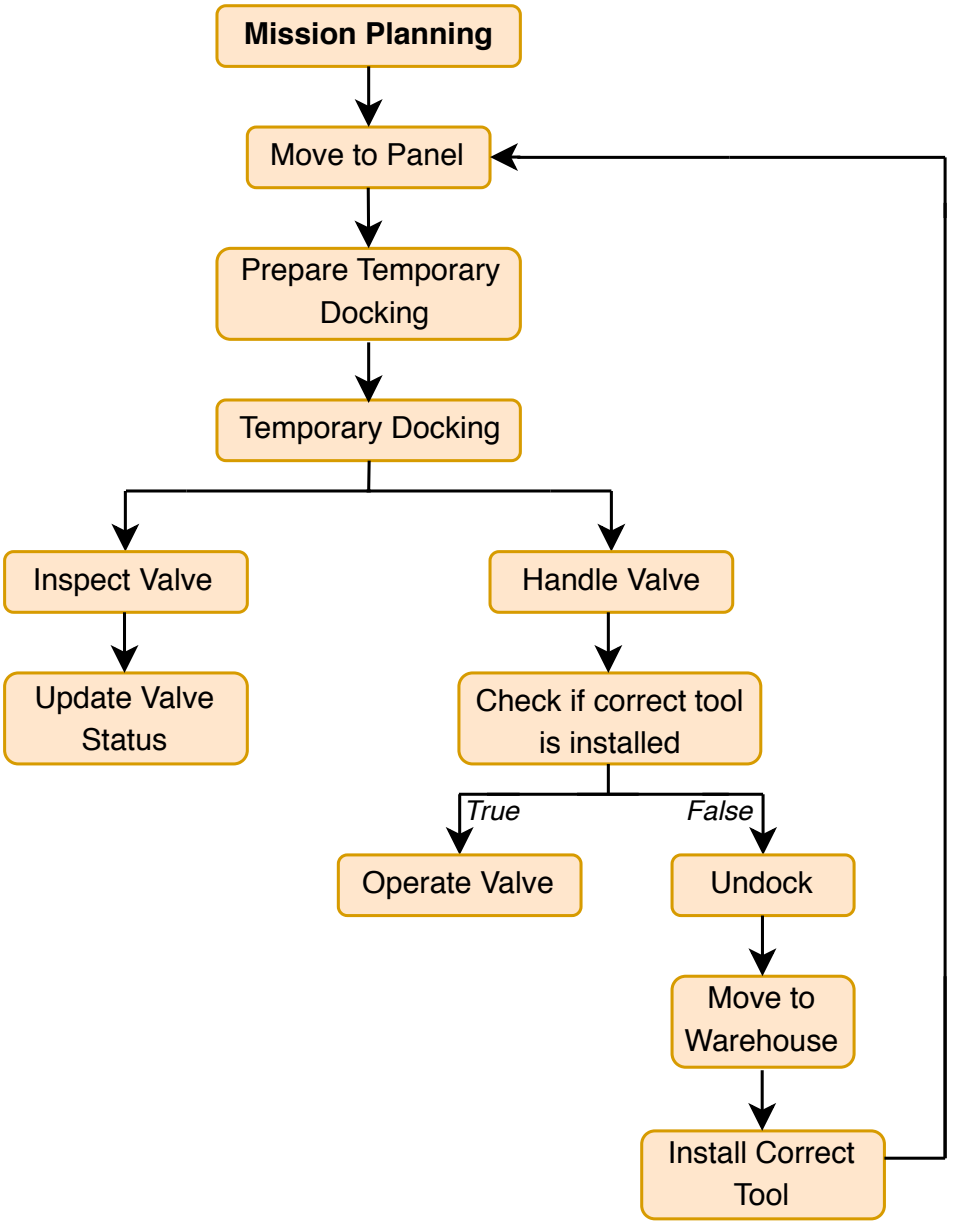


Figure 4.4: Defined mission illustrated as a flowchart

4.3 Simplifications and Assumptions

The defined mission planning problem aims to reflect a potential real IMR mission, while to some extent being solvable and explainable using the three methods introduced earlier in section 2.1.1.3, 2.1.2.1 and 2.2.2.1. The defined mission planning problem was still in need for more simplified assumptions to execute the mission and these will further be explained in this section.

For all the implemented scenarios, the UID always starts at the docking station. Furthermore, it is assumed that the UID is undocked when the mission starts. This means that the UID has been commanded by the operation center to operate the required valve, and it has consequently undocked from the docking station to prepare the execution of the mission. The aforementioned definition of the mission starts from this point.

For simplicity, there exists only one panel in the defined environment. This is seldom the case, as there would normally be numerous panels in one area of the subsea installation and the operation center has to specify which valve on which panel needs to be operated. Secondly, the UID is required to temporary dock close to the valves in order to either inspect or operate them. This is due to the fact that there is a low level of visibility caused by the combination of darkness and marine growth in the deep of the ocean, as explained by the ROV operators from Oceaneering.

The structure of the mission planning problem has also been based on how the panel is designed. As mentioned previously, the UID is required to temporary dock to the panel in order to manipulate, i.e. inspect or operate, the valves. Due to this requirement, the panel is designed accordingly and contains three bars to which the UID can temporary dock to while manipulating the valves. Another assumption is that for each bar, the UID can access two valves simultaneously. Figure 4.5 illustrates the design of the panel consisting of six valves. Furthermore, the pair of valves that are accessible from each of the bars are considered to be manipulable using the same tool. For instance, if one assumes that valves 1 and 4 constitute one pair, then both of the valves are manipulable by e.g. the manipulator torque multiplier tool.

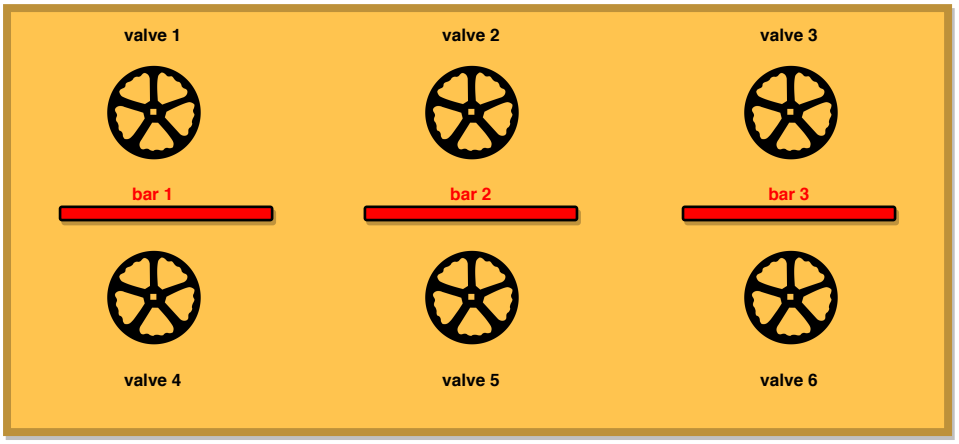


Figure 4.5: Structure of the subsea panel in the planning problem

Lastly, the task of manipulating, i.e. handling/operating, a valve is initially a very complex task. Due to the complexity of the task, it is generally divided into multiple smaller tasks in order to actually execute it. In the works of this thesis, it is assumed that manipulating a valve is only one single task.

The essential simplifications and assumptions related to the mission may be summed up as following:

- The mission always starts with the undocked UID at the docking station
- There is only one panel at the subsea installation
- The UID can access two valves simultaneously when temporary docked to the panel
- Manipulating the valve is considered one single task
- Each pair of valves is manipulated by using the same tool

4.4 Problem Formulated in PDDL

Finally, the mission planning problem is described using the modeling language PDDL, as this is the de facto standard for defining planning problems for AI planning methods. Listing 4.1, 4.2, 4.3, 4.4 and 4.5 give an overview of how the planning problem is formulated, as an important aspect of this thesis is how to formulate such a mission planning problem.

Listing 4.1 uses PDDL as to express the domain information, including the types, predicated and objects in the defined environment. Furthermore, listing 4.2 expresses both the initial and goal states for the mission planning problem as previously described in section 4.2. Both the initial and goal state contains a requirement denoted in italic; "*correct-tool-installed*". The italic font represents that this requirement is set to either "correct-tool-installed" of "**not** correct-tool-installed" which is dependent on what is assumed beforehand. Furthermore, listing 4.3 and 4.4 describe all the actions related to valve manipulation and docking, respectively. Finally, listing 4.5 describes the actions related to movement of the UID, in addition to change of tool.

```
(define (problem UID Mission Planning)
    (:domain subsea installation)
    (:types
        location
        valve
        tool
        bar
    )
    (:predicates
        (at ?l - location)
        (valve_needs_inspection ?v - valve)
        (valve_checked ?v - valve)
        (valve_needs_handling ?v - valve)
        (valve_operated ?v - valve)
        (correct_tool_installed ?v - valve ?t - tool)
        (temporary_docked ?b - bar)
        (ready_to_finaldock  ?l - location)
        (ready_to_tempdock ?l - location)
        (finished_mission)
        (final_docked)
    )
    (:objects
        docking station, panel, warehouse - location
        v1, v2, v3, v4, v5, v6 - valve
        t1, t2, t3 - tool
    )
```

Listing 4.1: PDDL description of the defined planning problem's domain

```
(:init
    at(docking station) and
    valve_needs_handling(v1) and
    valve_needs_inspection(v2) and
    valve_needs_inspection(v3) and
    valve_needs_inspection(v4) and
    valve_needs_inspection(v5) and
    valve_needs_inspection(v4) and
    correct-tool-installed(v1) or not correct-tool-installed(v1) and
    not finished_mission )
(:goal
    not at(docking station) and
    not valve_needs_handling(v1) and
    not valve_needs_inspection(v2) and
    not valve_needs_inspection(v3) and
    not valve_needs_inspection(v4) and
    not valve_needs_inspection(v5) and
    not valve_needs_inspection(v4) and
    not correct-tool-installed(v1) or correct-tool-installed(v1) and
    finished_mission
)
```

Listing 4.2: PDDL description of the defined planning problem's initial and goal state

```
(:action inspect valve
    :parameters(?v - valve
                ?b - bar
                ?panel - location)
    :precondition(at(panel) and
                  temporary_docked(b) and
                  valve_needs_inspection(v))
    :effect(not valve_needs_inspection(v))
)
(:action operate valve
    :parameters(?v - valve
                ?b - bar
                ?panel - location)
    :precondition(at(panel) and
                  temporary_docked(b) and
                  valve_needs_handling(v))
    :effect(not valve_needs_handling(v))
)
```

Listing 4.3: PDDL description of the defined planning problem's actions

```
    (:action prepare temporary docking
        :parameters(?v - valve
                    ?b - bar
                    ?panel - location)
        :precondition(at(panel) and
                      not temporary_docked and
                      valve_needs_handling(v) or
                      valve_needs_inspection(v) and
                      ready_to_tempdock)
        :effect(temporary_docked)
    )
    (:action undock
        :parameters(?panel - location
                    ?)
        :precondition()
        :effect(not temporary_docked and
                not ready_to_tempdock)
    )
    (:action temporarily dock
        :parameters(?v - valve
                    ?b - bar
                    ?panel - location)
        :precondition(at(panel) and
                      not temporary_docked and
                      valve_needs_handling(v) or
                      valve_needs_inspection(v))
        :effect(temporary_docked)
    )
    (:action prepare final docking
        :parameters(?dockingstation - location)
        :precondition(at(dockingstation and
                      finished_mission and
                      not ready_to_finaldock and
                      not final_docked))
        :effect(ready_to_finaldock)
    )
    (:action final dock
        :parameters(?dockingstation - location)
        :precondition(at(dockingstation) and
                      finished_mission and
                      ready_to_finaldock and
                      not final_docked)
        :effect(final_docked)
    )

)
```

Listing 4.4: PDDL description of the defined planning problem's docking-related actions

```
(:action change tool
    :parameters(?t -tool
                ?warehouse - location
                ?v - valve)
    :precondition(at(warehouse)
                  not correct-tool-installed(v, t))
    :effect(correct-tool-installed(v, t))
)
(:action move
    :parameters(?l1 - location
                ?l2 - location)
    :precondition(at(l1))
    :effect(at(l2))
)
```

Listing 4.5: PDDL description of the defined planning problem's actions

4.5 Implementation

As mentioned previously in the introduction, the work of this thesis includes methods from both AI Planning and Reinforcement Learning. The theory presented in chapter 2 introduced the reader to three methods within AI planning; STRIPS, HTN and Graphplan, and one method within Reinforcement learning. Despite of this, the thesis include only the implementation of Graphplan and HTN from AI planning and Q-learning from Reinforcement Learning.

The STRIPS planner is excluded from the scope of this thesis, as the Graphplan algorithm is preferred in solving the defined mission. The main reason for this choice is that Graphplan is able to overcome the problem with the STRIPS planner and the occurrence of the Sussman anomaly, as introduced in section 2.1.1.2, which might occur when solving this mission planning problem. For instance, re-visiting the same location, such as the panel, due to for example the incorrect tool being installed might potentially invoke the Sussman anomaly.

The first implementations to be explored are the AI planning methods with HTN and Graphplan. The structure of dividing the mission into tasks, which is how HTN solves a planning problem, will be exploited for implementing and solving the previously introduced mission planning problem for Graphplan. Therefore, the implementation of HTN is explored first and thereby followed by the implementation of Graphplan. The implementation of the Reinforcement Learning algorithm, i.e. Q-learning, builds on some of the aspects in Graphplan with regard to the used states and action, and is therefore described at the end.

4.5.1 Hierarchical Task Network

As described previously in section 2.1.2.1, a planning problem which is solved using HTN must be structured as tasks. Consequently, the introduced mission planning problem is defined by using the task structure. The mission planning problem is divided into three parts, which represents the main aspects of the mission. These are: ”Handle valve”, ”Inspect valve” and ”Dock to station”. Hence, the main task to be executed is ”Mission Planning” and the sub-tasks are ”Handle valve”, ”Inspect valve” and ”Dock to station”. It is also worth mentioning that the UID’s initial state remains as stated in the mission definition.

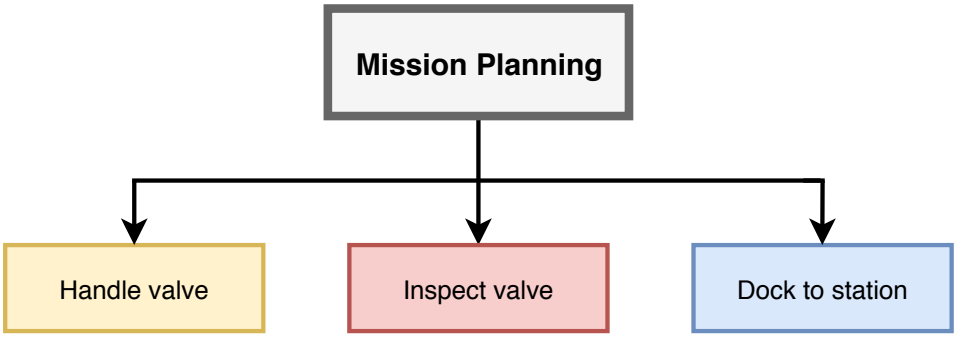


Figure 4.6: Overall description of the main tasks of the mission planning problem for the HTN method

From the mission definition and problem formulation, the first part of the defined mission planning problem is to operate valve 1 on the panel. In order to execute this task, the UID has to move from the docking station to the panel with the valves. Subsequently, the UID has to prepare itself for the temporary docking to the panel before actually docking to it. Upon temporary docking, the UID has to verify if the correct tool is installed before operating the valve. Depicted as a flowchart, figure 4.7 illustrates the task of handling the valve.

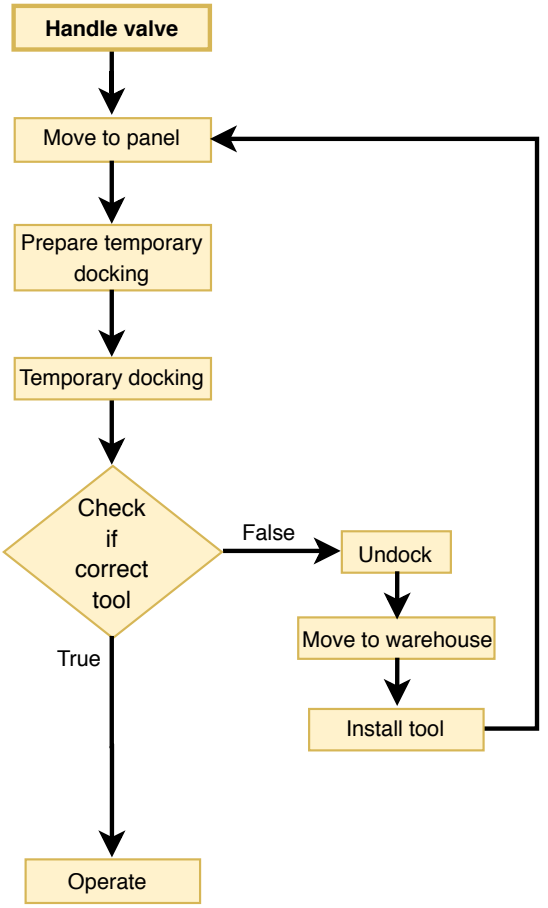


Figure 4.7: Overall description of the main tasks of the mission planning problem for the HTN method

The second part of the defined mission planning problem is to inspect the remaining of the valves on the panel. A flowchart describing this task is illustrated in figure 4.8. At this stage, the UID is already at the panel since the task is performed subsequently to the ”Handle valve” task. Regardless, the UID verifies that it is indeed at the panel as it might harm the UID or installation to perform actions when not in the correct state. For instance, preparing to dock or try to temporary dock when the UID is already docked to the docking station does not make any sense.

As previously explained in the mission planning problem definition, the UID has to inspect five valves in addition to operate one. In order to achieve this, the UID has to undock from the current bar and dock to the subsequent bar. Therefore, the dotted line in figure 4.8 indicates which methods or operators must be repeated in order to reflect the inspection of the remaining valves. Upon inspecting the last valve, the UID stays temporarily docked to the panel until the next task is invoked.

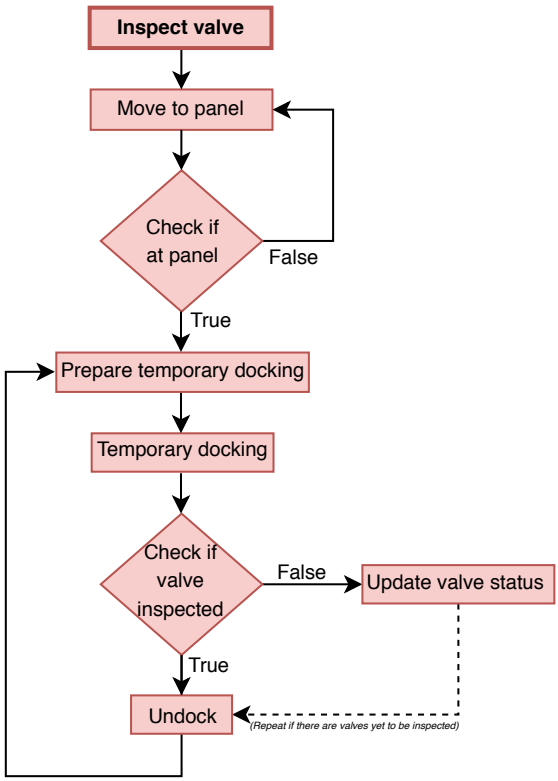


Figure 4.8: Overall description of the main tasks of the mission planning problem for the HTN method

Finally, the last part of the defined mission planning problem is to dock to the docking station. This is performed upon completion of the mission related to the valve, which includes both inspection and manipulation. As a starting point the UID verifies if it has completed the mission. If it has indeed completed the inspection and operation of valve as defined by the mission planning problem, it proceeds to undock from the panel and move to the docking station. The UID prepares to dock and thereby performs the actual docking to the docking station.

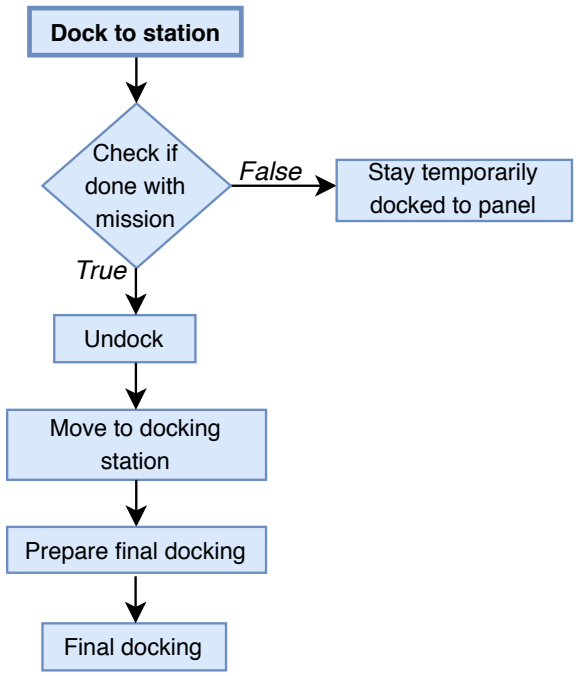


Figure 4.9: Overall description of the main tasks of the mission planning problem for the HTN method

In order to perform all the aforementioned tasks for the mission planning problems, they are broken down as methods and operators as explained in section 2.1.2.1. Table 4.1 gives an overview of the implemented methods and operators, which reflects if a task is primitive or non-primitive.

Methods	Operators
mission planning	move
inspect panel	prepareTempDocking
check valves	tempDock
handle valve	undock
operateValve	installTool
dock to docking station	operateValve
	updateValveStatus
	prepareFinalDocking
	dockToStation

Table 4.1: Overview of the defined methods and operators for HTN

4.5.2 Graphplan

The implementation of the Graphplan algorithm differs from the implementation of HTN, as the approaches are quite different. Instead of using tasks, the mission planning problem is defined using states and actions. Therefore, the mission planning problem's implementation is to some extent similar to the mission definition description in PDDL in listing 4.1, where the states are represented by the predicates.

Some inspiration was also drawn from the HTN method when implementing the Graphplan method for the same planning problem by dividing the problem into multiple sub-problems. In a similar manner, the planning problem is divided into one part for the ”valve handling”, one part for the ”valve inspection” and a part for ”docking to station”. Additionally, the aspect of re-docking from one bar to the next is considered a sub-problem as well. The main reason for this division is that the state-space of the mission planning problem becomes quite large when there are ten actions and twelve states. Furthermore, by limiting the size of the problem one increases the chances of convergence and consequently find a reasonable solution.

Otherwise, the domain for the Graphplan algorithm is similar to the one described for the general mission definition in listing 4.1. The actions and their respective precondition on the contrary differs a bit from the general mission definition. The main reason for this is that the used library for the implementation required some additional constraints on the precondition for each action to ensure that only one action could be taken in some state. A consequence of not having additional constraint is that the algorithm becomes inconclusive in solving the defined mission planning problem.

In addition, the predicates in the Graphplan algorithm are defined quite similarly as one would do for the STRIPS method by using first-order logic as explained in section 2.1.1.1. This means that the predicates can be described as booleans, i.e. by either ”true” or ”false”.

An overall description of the states and actions for the defined mission planning problem is illustrated in figure 4.10. The red circles represent the location-related states, while the grey circles represent the remaining states. The arrows indicate the actions that make the UID transition from one state to another.

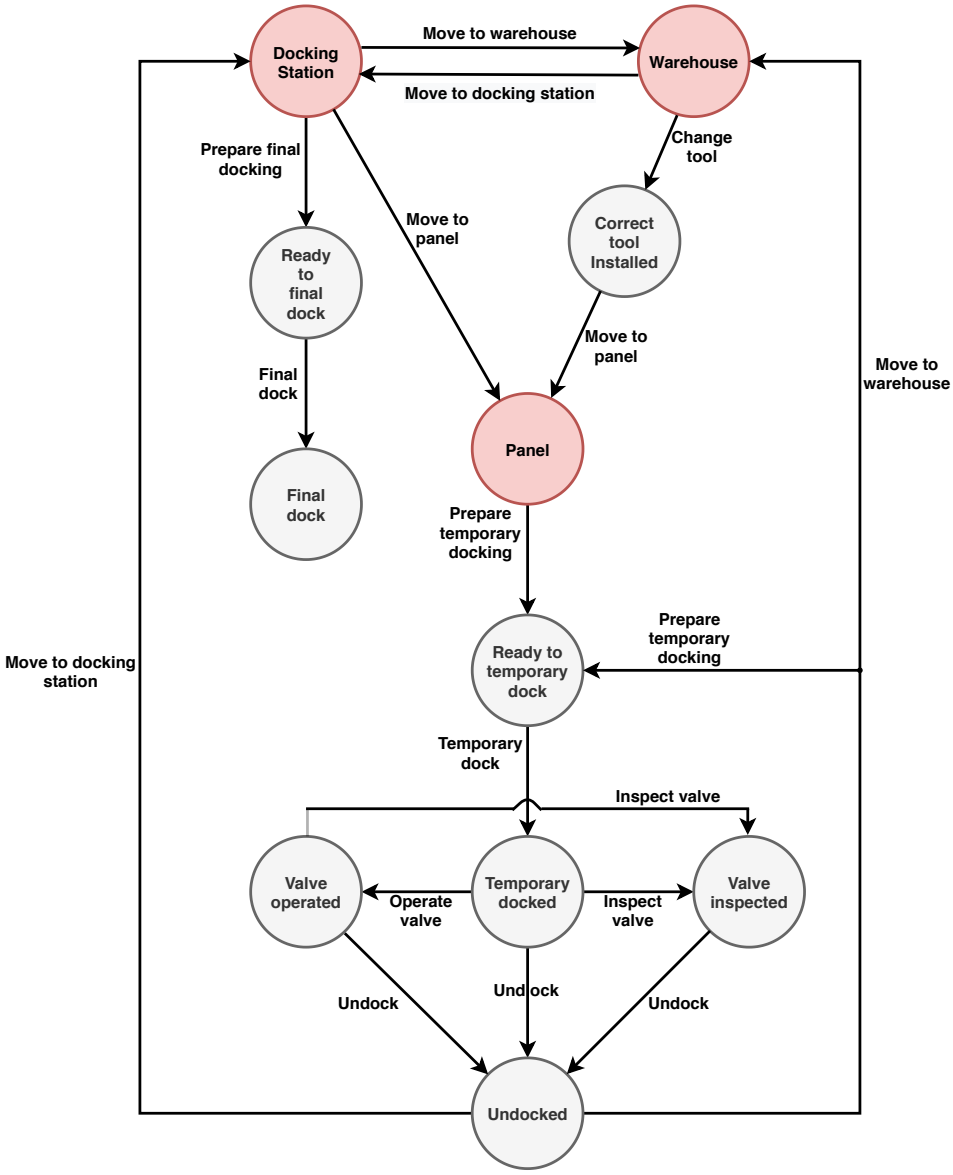


Figure 4.10: Overall state-space of the subsea installation used in the planning problem

In order to define the mission planning problem as three sub-missions, it is necessary to define both initial and goal states to represent where one sub-mission ends and the subsequent sub-mission starts. These are defined in listing 4.6, 4.7, 4.8 and 4.9, respectively. It is worth mentioning that the aforementioned listings do not follow the rules of modeling with PDDL, and are rather just on a generalized format.

```
init: At(DockingStation) and
      not At(Panel) and
      not At(Warehouse) and
      not ReadyToTempDock and
      not ValveOperated and
      not TempDocked and
      CorrectToolInstalled and
      not FinishedMission and
      not PanelVisited and

goal: ValveOperated
```

Listing 4.6: Initial and goal state for the valve operation part of the Graphplan implementation

The second part of the mission represents the ”valve inspection”-aspect of the mission planning problem. The initial and goal state for this part is described in listing 4.7.

```
init: ValveOperated and
      At(Panel) and
      not At(Warehouse) and
      not At(DockingStation) and
      TempDocked and
      not PanelVisited and
      not ValveInspected

goal: ValveInspected
```

Listing 4.7: Initial and goal state for the valve inspection part of the Graphplan implementation

The re-docking aspect of the mission essentially undocks the UID from the current bar and docks to the subsequent bar. Upon temporary docking to the new bar, the UID inspects the valves that are accessible. After inspecting the pair of accessible bars, the UID proceeds to the next bar. The initial and goal state for this part is described in listing 4.8.

```
init: TempDocked and
      At(Panel) and
      not At(Warehouse) and
      not At(DockingStation) and
      not PanelVisited and
      not ValveInspected

goal: TempDocked and
      PanelVisited
```

Listing 4.8: Initial and goal state for the re-docking part of the Graphplan implementation

The last sub-mission is to dock to the docking station upon completion of inspection and manipulating all the valves at the panel. The initial and goal states are described in listing 4.9.

```
init: not At(DockingStation) and
      At(Panel) and
      not At(Warehouse) and
      FinishedMission and
      ValveOperated and
      PanelInspected and
      TempDocked and
      not ReadyToFinalDock

goal: FinalDocked
```

Listing 4.9: Initial and goal state for the final docking part of the Graphplan implementation

In order to perform all the aforementioned sub-missions, it is necessary to to define the actions that are valid within the mission planning domain. In total, twelve actions are defined for the Graphplan algorithm. They are all described in listing 4.10 with their corresponding preconditions and effects.

```
action: "MoveToPanel"
precondition = not At(Panel) and
               not ValveOperated and
               not TempDocked
effect = At(Panel) and
         not At(DockingStation) and
         not At(Warehouse)
```

```
action: "MoveToWarehouse"
precondition = not CorrectToolInstalled and
               not At(Warehouse) and
               ValveOperated and
               not TempDocked and
               PanelVisited
effect = At(Warehouse) and
         not At(Panel) and
         not At(DockingStation)

action: "MoveToDocking"
precondition = not At(DockingStation) and
               FinishedMission and
               not TempDocked
effect = At(DockingStation) and
         not At(Panel) and
         not At(Warehouse)

action: ReadyToTempDock
precondition = not ReadyToTempDock and
               At(Panel) and
               not At(Warehouse) and
               not At(DockingStation) and
               not TempDocked
effect = ReadyToTempDock

action: TempDock
precondition = ReadyToTempDock and
               At(Panel) and
               not At(Warehouse) and
               not At(DockingStation) and
               not PanelVisited
effect = TempDocked and
         PanelVisited and
         not ReadyToTempDock

action: Undock
precondition = TempDocked and
               At(Panel) and
               not At(Warehouse) and
               not At(DockingStation)
effect = not TempDocked  and
         not ReadyToTempDock
```

```
action: OperateValve
precondition = At(Panel) and
               not At(DockingStation) and
               not At(Warehouse) and
               TempDocked and
               not ValveOperated and
               CorrectToolInstalled
effect = ValveOperated and
         not CorrectToolInstalled

action: InspectValve
precondition = not ValveInspected and
                 TempDocked and
                 At(Panel) and
                 not At(DockingStation) and
                 not At(Warehouse)
effect = ValveInspected

action: ChangeTool
precondition = At(Warehouse) and
                 not At(Panel) and
                 not At(DockingStation) and
                 not CorrectToolInstalled and
                 PanelVisited
effect = CorrectToolInstalled

action: PrepareFinalDocking
precondition = At(DockingStation) and
                 not At(Panel) and
                 not At(Warehouse) and
                 not ReadyToFinalDock
effect = ReadyToFinalDock

action: FinalDock
precondition = At(DockingStation) and
                not At(Panel) and
                not At(Warehouse) and
                ReadyToFinalDock
effect = FinalDocked
```

Listing 4.10: Graphplan implementation with the defined actions

4.5.3 Q-learning

The Q-learning algorithm requires the implementation of a custom environment which would reflect the defined mission planning problem and its corresponding domain. The domain has to contain both states and actions which the software agent will exploit in order to learn about its surroundings. Consequently, the domain for the subsea mission planning problem is designed in a similar manner as the one for the Graphplan algorithm, as this ensures that the states and actions are nearly the same. An overview of the domain of the mission planning problem is depicted in figure 4.10.

The main difference between the mission planning problem for the Q-learning algorithm compared to the AI Planning methods is that the mission planning problem is simplified. First and foremost, it is assumed that the panel in the defined domain only contains two valves. The first valve needs to be operated and the second valve needs to be inspected. This simplification is done due to the difficulty of developing a reward function such that the software agent learns to repeat the same sequence of actions a certain amount of times. For instance, if there are six valves on the panel the UID has to operate valve 1, inspect valve 2, then re-dock to next bar in order to access the next two bars and finally repeat the same actions for re-docking to the last bar and inspecting the remaining valves.

The overall mission planning problem is divided into three parts, in a similar manner as for the AI Planning methods. The first sub-mission is to operate valve 1, followed by inspecting valve 2 and finally docking to the docking station upon completion. However, the domain of mission planning problem still contains the states and actions related to the change of tool and re-docking. By keeping these aspects in the overall mission planning domain, the complexity of the overall mission is increased as the software agent has the option to explore these state even though they are not a part of the desired objective. Figure 4.11 illustrates the state-space representation assumed in the implementation of the Q-learning algorithm. The states denoted in bold are the initial state/goal states for each of the sub-missions and table 4.2 also gives an overview of the initial and goal states for each sub-mission.

	initial state	**goal state**
sub-mission 1	”Docking Station”	”Valve Operated”
sub-mission 2	”Valve Operated”	”Valve Inspected”
sub-mission 3	”Valve Inspected”	”Docking Station”

Table 4.2: The initial and goal states for each of the sub-missions for the Q-learning algorithm

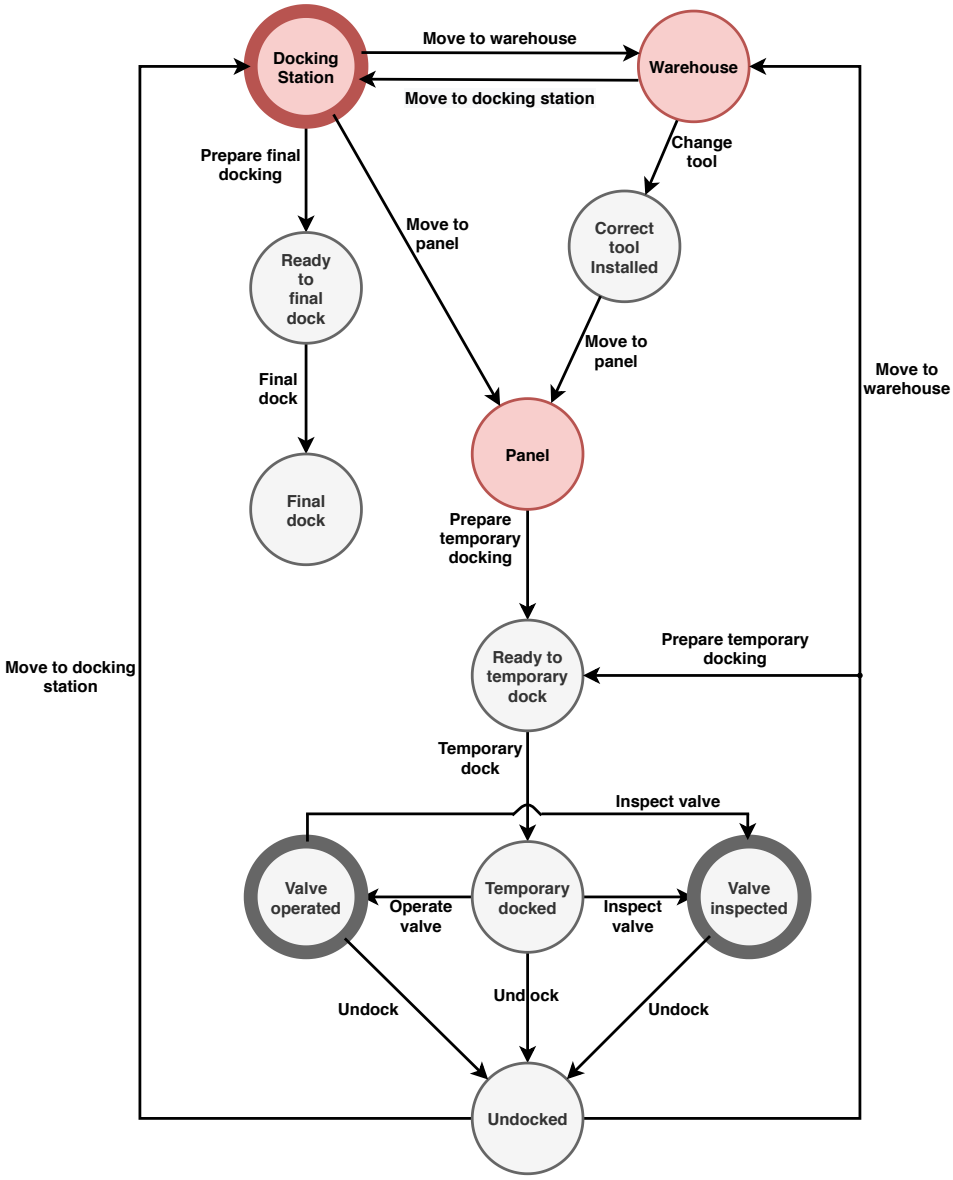


Figure 4.11: Modified state-space representation of the mission planning problem domain for Q-learning

The implementation of the custom Reinforcement Learning Environment, which is utmost necessary for expressing the mission planning problem in such a manner that the software agent is able to learn the environment by exploring it, requires several modules. First and foremost, the observation and action spaces had to be determined. The observation space contains all the valid observations the environment, i.e. all states that are considered as valid observation by the software agent. In the same manner, the action space contains all the objects that are considered as valid actions in the defined environment. These spaces are defined based on figure 4.11.

Both the action and observation spaces were defined as discrete, i.e. as Boolean variables, meaning that they are either valid or not valid. Initially the discrete representation of the spaces denotes each of the object in the space with a number and the corresponding value of either ”True” of ”False”. To gain a better overview of each state or action and its corresponding value, the data structure of a dictionary was used to connect the discrete object to its corresponding name defined with the string datatype. Listing 4.11 given an overview of the defined observation and action spaces.

```
self.action_space = spaces.Discrete(11)
self.action_space_dictionary =
    {0 : "MoveToPanel", 1 : "MoveToWarehouse", 2 : "MoveToDocking",
     3 : "PrepareTempDock", 4 : "TempDock", 5 : "Undock",
     6 : "InspectValve", 7 : "OperateValve",
     8 : "ChangeTool", 9 : "PrepareFinalDocking", 10 : "FinalDock"}

self.observation_space = spaces.Discrete(11)
self.observation_space_dictionary =
    {0 : 'AtPanel', 1 : 'AtWarehouse', 2 : 'AtDockingStation',
     3 : 'ReadyToTempdock', 4 : 'TempDocked', 5 : 'Undocked',
     6 : 'ValveInspected', 7 : 'ValveOperated',
     8 : 'CorrectToolInstalled', 9 : 'ReadyToDock',
     10 : 'FinalDock'}
```

Listing 4.11: Observation and Action Spaces for the custom RL environment

Moreover, in order for the software agent to learn about the environment and consequently how to act within it, a reward function is defined to indicate to the software agent which behaviour within the environment is beneficial in the attempt of reaching the desired objective. Essentially, the implemented reward function for the defined mission planning problem is quite simple. For each state the software agent reaches which is not the objective, i.e. the goal state, the agent receives a penalty of -1. If the next state the agent reaches is the goal state, then it is rewarded $+10$.

Furthermore, the number of episodes is defined as 10000, in order to have a high number of training sequences. The probability of the software agent taking a greedy action is defined as $\epsilon = 0.7$, and is used by the ϵ-greedy method in order to determine which action to

take for each step in the current episode. Moreover the learning rate is defined as $\alpha = 0.1$ and the discount rate is set to $\gamma = 1.0$. The definition of these variables and the definition of the Q-learning algorithm can be found in section 2.2.2.1.

Another aspect which was taken into consideration was the fact that all actions were not valid from each of the states. For instance, if the UID is at the Warehouse it cannot inspect the valve. Consequently, after the ϵ-greedy selects an action the implemented algorithm checks if the action is valid based on the current state of the software agent. If it is valid, then the software agent can proceed to calculate the Q-value in the Q-table. On the contrary, if the the action is not valid, then the Q-value is not calculated and the Q-table remains unchanged.

Chapter 5

Results and Analysis

This chapter describes how the different methods are able to solve the previously defined mission planning problem. Furthermore, the AI planning methods are explainable and can therefore indicate how the algorithm proceeds to start at some initial state and reaches the desired goal or objective. The Reinforcement Learning method

5.1 Mission Planning Problem without Replanning

As a starting point this section includes the obtained results for the mission planning problem described in Chapter 4, however the UID does not need to change the tool in order to operate the given valve. The need of changing the manipulation tool is discovered mid-mission, i.e. when the UID is at the panel and ready to operate the valve, and is consequently considered a replanning.

5.1.1 Solution using Hierarchical Task Network

In the same manner as the definition of the mission planning problem in section 4.5.1, the obtained results are also divided in three to reflect the tasks. The first main task is ”handle valve” and the corresponding solution is described in table 5.1. The color-coordination used in the table represent the three different tasks, as depicted in figure 4.6, where yellow in the task ”handle valve”, red is ”inspect valve” and blue is ”dock to docking station”. The main task is ”mission planning” and is denoted in bold as this is the main starting task. Moreover, at each depth the chosen action is also denoted in bold. For instance, in depth 1 the current task is **handle valve**.

As table 5.1 represents the start of the mission, depth 0 only contains one task, which is mission planning. This task consists of three sub-tasks which become available of the algorithm upon exploration. The main task is replaced by the three sub-task; ”handle valve”,

”inspect panel” and ”dock to docking station”, which have to be executed in order for the main task to be executed. In depth 1, the first sub-task is selected which also contains new sub-tasks. These new tasks are added to the tasks the algorithm need to execute in order to solve the mission planning problem. For each depth, the first action in the ”Tasks”-column is selected and performed, until it reaches the last sub-task for the task ”handle valve” which is to operate the valve. This illustrates the occurrence of hierarchy in a task network.

Depth	Tasks	New Tasks
0	**mission planning**	handle valve inspect panel dock to docking station
1	**handle valve** inspect panel dock to docking station	move to panel prepareTempDocking tempDock operate valve
2	**move to panel** prepareTempDocking tempDock operate valve inspect panel dock to docking station	
3	**prepareTempDocking** tempDock operate valve inspect panel dock to docking station	
4	**tempDock** operate valve inspect panel dock to docking station	
5	**operate valve** inspect panel dock to docking station	

Table 5.1: The partial plan for obtained by solving the ”handle valve” task with HTN

The second sub-task is ”inspect panel”, denoted in red in table 5.2. The ”inspect panel” task consists of only one sub-task, i.e. ”inspect valve”. At this point, the UID is docked to the first bar as depicted in figure 4.5 after having operated the first valve. Therefore, the UID can inspect valve 2 and update its status without the need of re-docking.

In depth 9 the UID has to continue the inspection, but since it has already inspected the valves that are reachable from the bar it is currently docked to, it has to re-dock to the next bar. The re-docking requires the UID to undock, prepare for temporary docking and then temporary dock. In table 5.2 the depths $10 - 14$ are repeated in order to inspect all the valves on the defined panel. At depth 30, there are not any valves which have not already been inspected and therefore this sub-task has successfully been executed.

Depth	Tasks	New Tasks
6	**inspect panel** dock to docking station	inspect valve
7, 15, 23	**inspect valve** dock to docking station	updateValveStatus inspect valve
8, 16, 24	**updateValveStatus** inspect valve dock to docking station	
9, 17, 25	**inspect valve** dock to docking station	undock prepareTempDocking tempDock updateValveStatus inspect valve
10, 18, 26	**undock** prepareTempDocking tempDock updateValveStatus inspect valve dock to docking station	
11, 19, 27	**prepareTempDocking** tempDockupdateValveStatus inspect valve dock to docking station	
12, 20, 28,	**tempDock** updateValveStatus inspect valve dock to docking station	
13, 21, 29	**updateValveStatus** inspect valve dock to docking station	
14, 22, 30	**inspect valve** dock to docking station	

Table 5.2: The partial plan obtained by solving the ”inspect panel” task with HTN for the replanning mission

The last remaining task to be executed is ”dock to docking station”, as the handling and inspection of valves has been completed. The task ”dock to docking station” contains four sub-tasks. First and foremost, the UID has to undock since it is temporarily docked to the panel. Upon undocking, the UID moves to the docking station in order to complete solving the mission planning problem. Upon arrival at the docking station, the UID prepares to dock and finally docks to the station.

Depth	Tasks	New Tasks
31	**dock to docking station**	undock move to docking station prepareFinalDocking dockToStation
32	**undock** move to docking station prepareFinalDocking dockToStation	
33	**move to docking station** prepareFinalDocking dockToStation	
34	**prepareFinalDocking** dockToStation	
35	**dockToStation**	

Table 5.3: The partial plan obtained by solving the ”dock to docking station” task with HTN for the replanning mission

5.1.2 Solution using Graphplan

As described in the implementation of the Graphplan algorithm in section 4.5.2 the mission planning problem is divided into three sub-missions. Each of the missions are solved separately and the final solution is a composition of the solutions obtained by the sub-mission individually.

As mentioned in the previous chapter in section 4.3, which describes the simplifications and assumptions related to the mission planning problem, the UID starts at the docking station. The goal state of the first mission is "Valve operated", and the found solution is depicted in figure 5.1. The actions A which are taken to reach the goal state from the initial state are the black arrows from the "Docking station" to the "Valve operated". At the end of this section, table 5.4 describes the total solution where $A_0 - A_3$ corresponds to the solutions to this sub-mission.

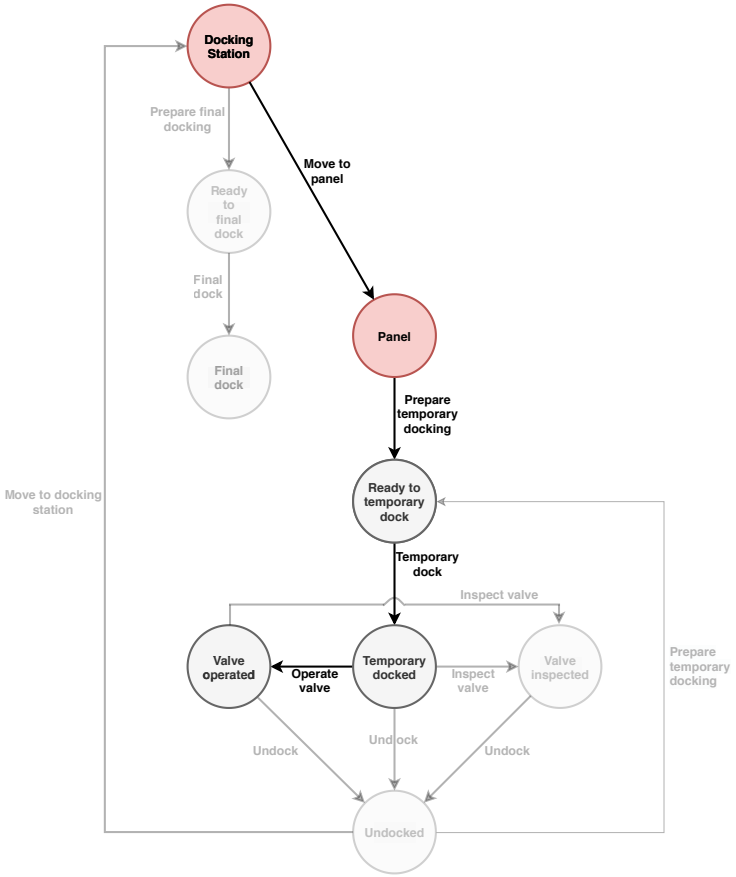


Figure 5.1: The obtained solution by the Graphplan algorithm for part one of the mission planning problem without replanning

The second part of the mission starts where the aforementioned sub-mission ends, i.e. ”Valve Operated”, and ends at ”Valve Inspected”. However, in order to consider the sub-mission as completed all the valves have to be inspected and therefore the algorithm expands the corresponding planning graph (see algorithm 2 in Chapter 2) for each valve which is yet to be explored. Since the UID is already docked to the panel and has operated valve 1, the UID can directly inspect valve 2 (see figure 4.5 for valve setup). Subsequently, it undocks from the current bar, prepares to temporary dock and docks to the next bar. From this bar it can inspect the two next valves. Lastly, it repeats the re-docking and inspects the two remaining valves which indicates the completion of the current sub-mission. In table 5.4, the actions $A4 - A_{14}$ describes the aforementioned actions for solving this part of the mission planning problem.

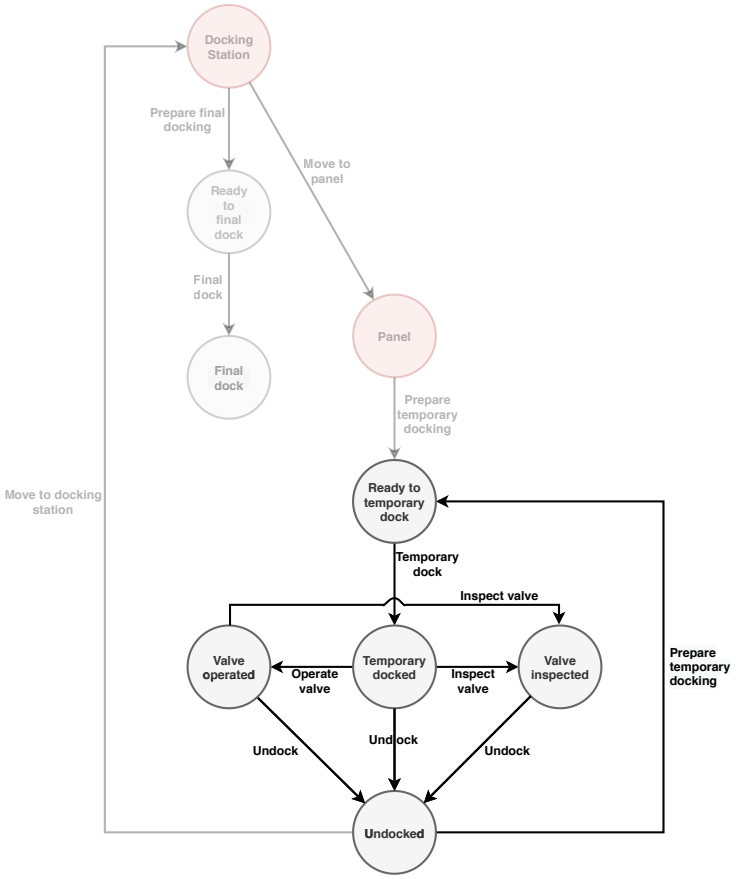


Figure 5.2: The obtained solution by the Graphplan algorithm for part two of the mission planning problem without replanning

The last part of the mission planning problem starts where the previous sub-mission ends, i.e. ”Valve Inspected”, and ends at the state ”Final dock”. The obtained solution by the Graphplan algorithm starts by undocking the UID from the bar on the panel, followed by moving to the docking station. Upon arrival at the docking station, the UID prepares to final dock before docking to the station. The solution for this sub-mission is represented in table 5.4 as actions $A_{15} - A_{18}$.

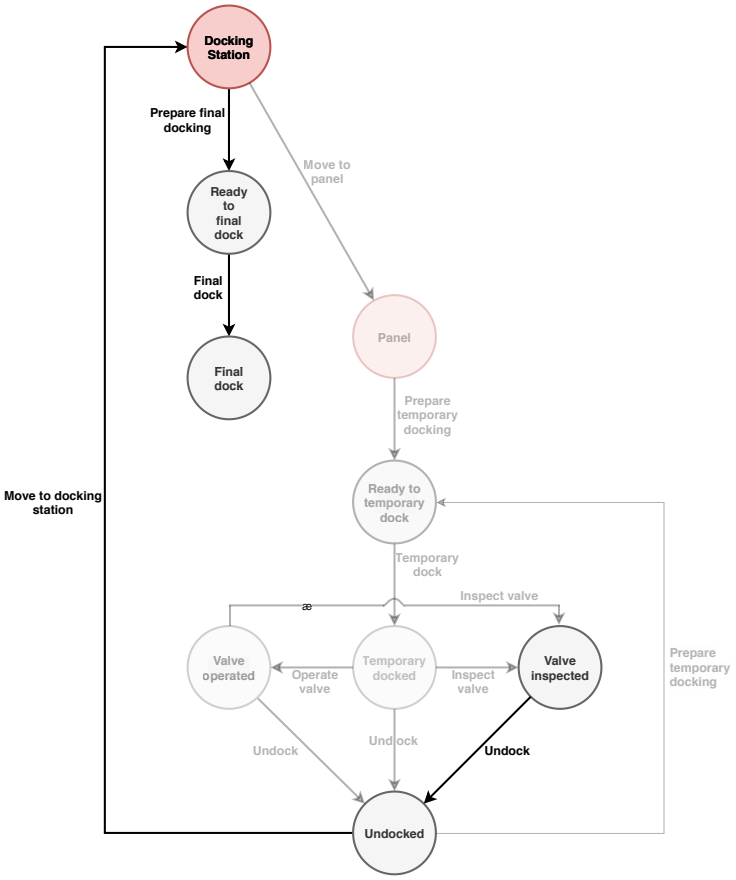


Figure 5.3: The obtained solution by the Graphplan algorithm for part one of the mission planning problem without replanning

Sub-mission	A	Action
1	$A_0 - A_3$	MoveToPanel ReadyToTempDock TempDock OperateValve
2	$A_4 - A_{14}$	InspectValve Undock ReadyToTempDock TempDock InspectValve InspectValve Undock ReadyToTempDock TempDock InspectValve InspectValve
3	$A_{15} - A_{18}$	Undock MoveToDocking PrepareFinalDocking FinalDock

Table 5.4: The overall solution obtained by the Graphplan algorithm for the mission planning problem without replanning

5.2 Mission Planning Problem with Replanning

The second scenario which is implemented in the work of this thesis includes an aspect of replanning. In terms of the defined mission planning problem this is implemented as the requirement of changing the tool on the UID, as it is not able to operate the valve without the correct tool installed.

5.2.1 Solution using Hierarchical Task Network

The overall solution for the mission planning problem with the replanning aspect is quite similar to the solution without the replanning aspect. Only the first task, i.e. ”handle valve”, differs in the overall solution and is described in table 5.5. The remaining two tasks, i.e. ”inspect panel” and ”dock to docking station” are described earlier in table 5.2 and table 5.3, respectively.

The depths $0-4$ in the solution of the ”handle valve” task for the mission planning problem with replanning is identical to the solution described earlier in table 5.1. The difference occurs at depth 5 when the UID is not able to operate the valve due to having the incorrect tool installed. Consequently, the execution of the task ”operate valve” is dependent on the

sub-task ”changeTool”. Depth 6 describes all the newly added tasks which are sub-tasks of ”changeTool”. The UID has to move to the warehouse from the panel, install the required tool, move back to the panel and thereby re-dock by preparing the temporary docking and finally temporary dock to the panel again. This is described in depths $7 - 11$ Upon change of tool, the UID is able to operate the valve and can continue the mission as described previously.

Depth	Tasks	New Tasks
0	**mission planning**	handle valve inspect panel dock to docking station
1	**handle valve** inspect panel dock to docking station	move to panel prepareTempDocking tempDock operate valve
2	**move to panel** prepareTempDocking tempDock operate valve inspect panel dock to docking station	
3	**prepareTempDocking** tempDock operate valve inspect panel dock to docking station	
4	**tempDock** operate valve inspect panel dock to docking station	
5	**operate valve** inspect panel dock to docking station	changeTool
6	**changeTool** operate valve inspect panel dock to docking station	move to warehouse install tool move to panel prepareTempDocking tempDock

Table 5.5 continued from previous page

Depth	Tasks	New Tasks
7	**move to warehouse** install tool move to panel prepareTempDocking tempDock operate valve inspect panel dock to docking station	
8	**install tool** move to panel prepareTempDocking tempDock operate valve inspect panel dock to docking station	
9	**move to panel** prepareTempDocking tempDock operate valve inspect panel dock to docking station	
10	**prepareTempDocking** tempDock operate valve inspect panel dock to docking station	
11	**tempDock** operate valve inspect panel dock to docking station	
12	**operate valve** inspect panel dock to docking station	
13	inspect panel dock to docking station	

Table 5.5: The partial plan obtained by solving the ”handle valve” task with HTN for the replanning mission

5.2.2 Solution using Graphplan

In the same manner as for the HTN method, the overall solutions obtained by the Graphplan algorithm for the mission planning problem with and without the replanning aspect are quite similar. The defined mission planning problem differs in the first sub-mission which is defined as going from the initial state of the ”Docking Station” and to the goal state of ”Valve operated”.

The first sub-mission starts identically to the first sub-mission for the mission planning problem without replanning, i.e. at the state of ”docking station” and moves to the panel in order to prepare the temporary docking before temporary docking to the panel. Furthermore, the UID tries to operate valve 1, but unlike in the previous mission the UID does not have the correct tool for operating the valve. Consequently, the UID needs to undock from the ”panel” and move to the ”warehouse” where it can change the manipulator tool. Upon completion, the UID moves back to the ”panel”, prepares the temporary docking and finally temporary docks to the same bar as previously. The UID has now the correct tool installed and is capable of operating the valve.

Figure 5.4 illustrates the visited states and the chosen actions for solving the mission planning problem. Furthermore, table 5.6 sums up the actions that are necessary for solving the mission planning problem, both for each sub-mission and the problem as a whole. The actions for the described sub-mission are $A_0 - A_9$, while the remaining actions are identical to the solutions for the previous planning planning problem which neglected the replanning aspect (see table 5.4). These actions are denoted as $A_{10} - A20$ and $A_{21} - A_{24}$ for sub-mission two and three, respectively.

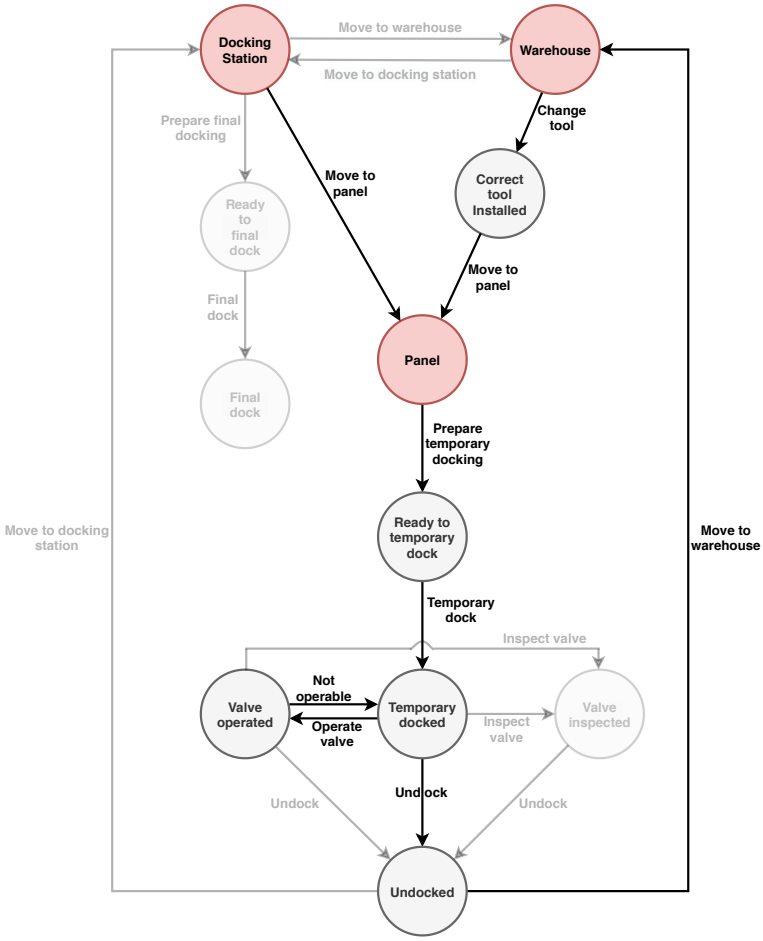

Docking Station
Move to warehouse
Warehouse
Move to docking station
Prepare final docking
Change tool
Move to panel
Correct tool Installed
Ready to final dock
Move to panel
Final dock
Panel
Final dock
Prepare temporary docking
Move to docking station
Ready to temporary dock
Move to warehouse
Temporary dock
Inspect valve
Not operable
Valve operated
Temporary docked
Valve inspected
Operate valve
Inspect valve
Undock
Undock
Undock
Undocked

Figure 5.4

Mission	A	**Action**
1	$A_0 - A_9$	MoveToPanel ReadyToTempDock TempDock Undock MoveToWarehouse ChangeTool MoveToPanel ReadyToTempDock TempDock OperateValve
2	$A_{10} - A_{20}$	InspectValve Undock ReadyToTempDock TempDock InspectValve InspectValve Undock ReadyToTempDock TempDock InspectValve InspectValve
3	$A_{21} - A_{24}$	Undock MoveToDocking PrepareFinalDocking FinalDock

Table 5.6: The overall solution obtained by the Graphplan algorithm for the mission planning problem with replanning

5.3 Mission Planning Problem solved with Q-learning

The results for the Q-learning algorithm differs from the implementation and consequently the results as well, since the assumed scenario for this method is a simplification of the defined mission planning problem as explained earlier in section 4.5.3. However, the simplified mission planning problem is also divided into three sub-missions which are quite similar to the sub-missions of the Graphplan algorithm. The three sub-missions for Q-learning are illustrated in figure 5.5, 5.6 and 5.7, respectively.

Furthermore, the first sub-mission is defined from the "Docking station" as initial state and "Valve operated" as goal state. Both these states are depicted as bold circles in figure 5.5. The corresponding Q-table for this sub-mission is denoted Q_1 and contains the results after completing the 10000 episodes which were defined for the Q-learning algorithm. Based on the Q-table, i.e. Q_1, one can determine which action is most beneficial in each of the defined states. The most beneficial action can be determined as rows in the Q-table represents each of the states in the environment and the columns represent the actions. The Q-learning algorithm chooses the action which maximizes the Q-value as explained in section 2.2.2.1, and these are summed up in table 5.7.

$$Q_1 = \begin{bmatrix} 0. & 9.6 & 9.6 & 9.8 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 9.7 & 0. & 9.6 & 0. & 0. & 0. & 0. & 0. & 9.6 & 0. & 0. \\ 9.7 & 9.6 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 9.4 & 0. \\ 0. & 0. & 0. & 0. & 9.9 & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 9.9 & 9.6 & 10. & 0. & 0. & 0. \\ 0. & 9.6 & 9.6 & 9.80. & 0. & 0. & 0. & 0. & 0. & 0. & \\ 0. & 0. & 0. & 0. & 0. & 9.7 & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 9.7 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 9.5 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \end{bmatrix}$$

State	Recommended action
'AtPanel'	PrepareTempDock
'AtWarehouse'	MoveToPanel
'AtDockingStation'	MoveToPanel
'ReadyToTempdock'	TempDock
'TempDocked'	OperateValve
'Undocked'	PrepareTempDock
'ValveInspected'	Undock
'ValveOperated'	MoveToPanel
'CorrectToolInstalled'	MoveToPanel
'ReadyToDock'	FinalDock
'FinalDock'	-

Table 5.7: Recommended actions based on the obtained Q-table for sub-mission 1

Based on table 5.7, if the agent always follows the recommended action, it will be able to reach the goal state in an efficient manner. For instance, by starting at the ”docking station” and following the recommended actions at each state result in the plan illustrated in figure 5.5. However, it is noteworthy that the states ”Ready to final dock” and final dock do not contribute in reaching the defined goal state, as each of the state only have one possible action.

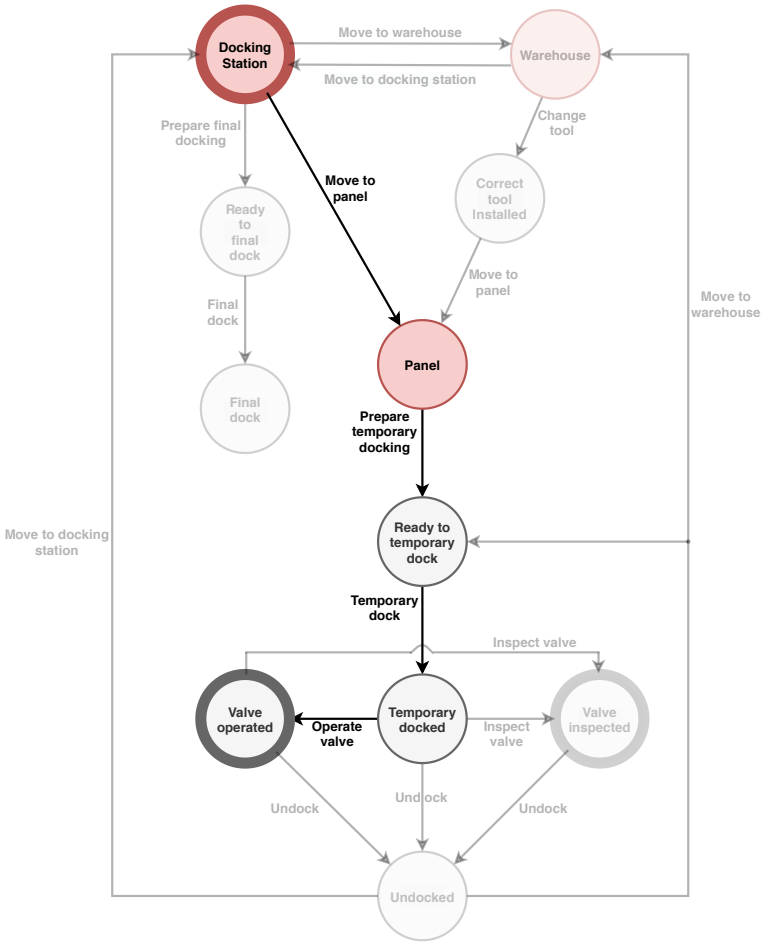


Figure 5.5: Overview of sub-mission 1 for the Q-learning algorithm

The second sub-mission starts at the state "Valve operated" and the desired goal state is "Valve inspected". These states are indicated in bold in figure 5.6, while the corresponding Q-table is denoted Q_2. As for sub-mission 1, the Q-table is the result of the Q-learning algorithm looping through the 10000 episodes in order to determine which actions the agent should perform at each of the states. Based on the Q-table Q_2, table 5.8 is derived as the recommended actions.

$$Q_2 = \begin{bmatrix} 0. & 9.6 & 9.6 & 9.8 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 9.7 & 0. & 9.6 & 0. & 0. & 0. & 0. & 0. & 9.6 & 0. & 0. \\ 9.7 & 9.6 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 9.4 & 0. \\ 0. & 0. & 0. & 0. & 9.9 & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 9.9 & 10. & 9.6 & 0. & 0. & 0. \\ 0. & 9.6 & 9.6 & 9.8 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 9.7 & 0. & 0. & 0. & 0. & 0. \\ 9.7 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 9.5 \\ 0. & 0. & 9.6 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \end{bmatrix}$$

State	Recommended action
'AtPanel'	PrepareTempDock
'AtWarehouse'	MoveToPanel
'AtDockingStation'	MoveToPanel
'ReadyToTempdock'	TempDock
'TempDocked'	InspectValve
'Undocked'	PrepareTempDock
'ValveInspected'	MoveToPanel
'ValveOperated'	Undock
'CorrectToolInstalled'	MoveToPanel
'ReadyToDock'	FinalDock
'FinalDock'	-

Table 5.8: Recommended actions based on the obtained Q-table for sub-mission 2

Figure 5.6 illustrates the solution based on the recommended actions obtained from the Q-table Q_2 and rendered in table 5.8. The solutions is based on whichever actions result in the highest reward and thereby chooses the solution with the fewest steps as each step to a state which is not the desired goal state give a penalty of -1.

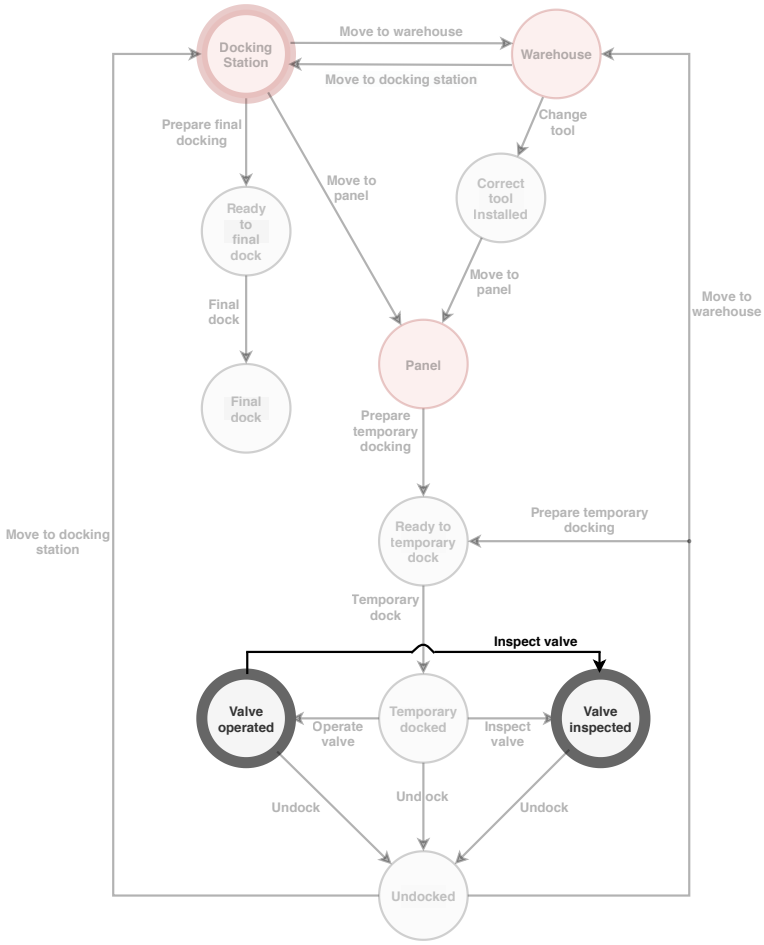


Figure 5.6: Overview of sub-mission 2 for the Q-learning algorithm

The third sub-mission is defined by the initial state "Valve Inspected" and goal state "Final dock", which are both denoted in bold in figure 5.7. This is considered the last part of the overall mission planning problem. The corresponding Q-table which is obtained after 10000 episodes, is denoted Q_3. Based on the Q-table Q_3, the recommended actions for each state can be extracted. Table 5.9 gives an overview of these recommended actions.

$$Q_3 = \begin{bmatrix} 0. & 9.7 & 9.8 & 9.399 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 9.7 & 0. & 9.8 & 0. & 0. & 0. & 0. & 0. & 9.6 & 0. & 0. \\ 9.7 & 9.7 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 9.9 & 0. \\ 0. & 0. & 0. & 0. & 9.499 & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 9.599 & 8.896 & 0. & 0. & 0. \\ 0. & 9.536 & 9.8 & 9.183 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 9.7 & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 9.57 & 0. & 0. & 0. & 0. & 0. \\ 9.7 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 10. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \end{bmatrix}$$

State	Recommended action
'AtPanel'	MoveToDocking
'AtWarehouse'	MoveToDocking
'AtDockingStation'	PrepareFinalDocking
'ReadyToTempdock'	TempDock
'TempDocked'	InspectValve
'Undocked'	MoveToDocking
'ValveInspected'	Undock
'ValveOperated'	Undock
'CorrectToolInstalled'	MoveToPanel
'ReadyToDock'	FinalDock
'FinalDock'	-

Table 5.9: Recommended actions based on the obtained Q-table for sub-mission 3

Figure 5.7 illustrates the solution based on the recommended actions obtained from the Q-table Q_3 and rendered in table 5.9. It is worth mentioning that the highlighted states and actions correspond to a solution which would be beneficial for the software agent as it maximizes the reward from the initial state to the goal state.

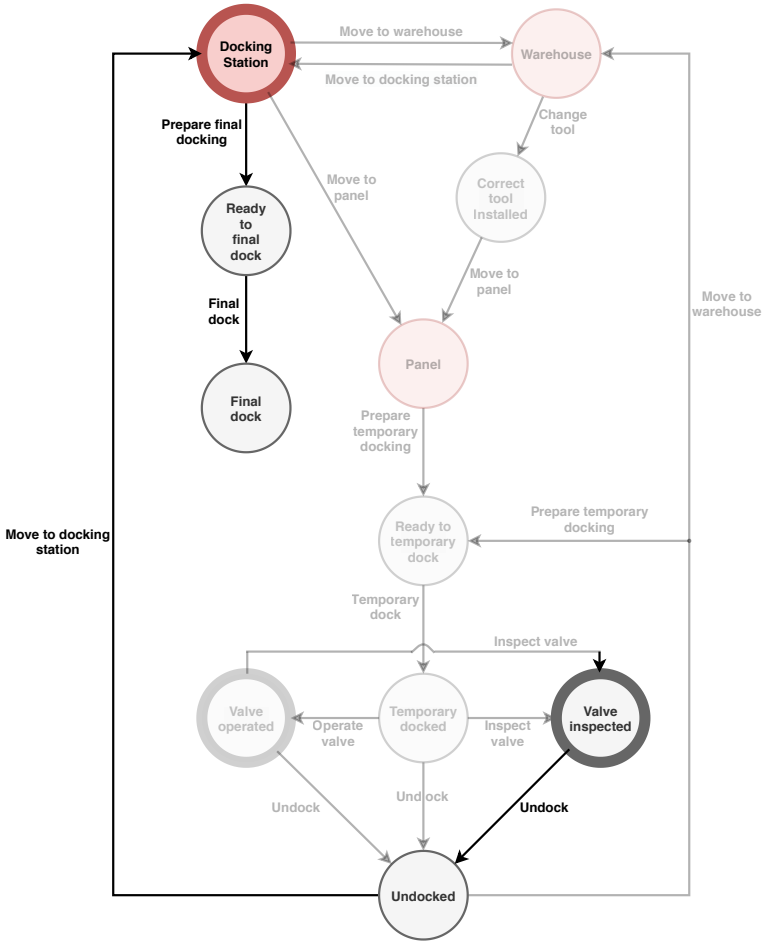


Figure 5.7: Overview of sub-mission 3 for the Q-learning algorithm

5.4 Analysis

First and foremost, it is worth mentioning that the three implemented methods are quite different in their structure and how the solve the mission planning problem. Secondly, the definition of the mission planning problem varies between the AI Planning methods and the RL method. However, some comparisons and resulting analysis can still be made.

The AI Planning methods, i.e. Graphplan and HTN, could both be compared with respect to their runtime as they solved the same mission planning problem. Both methods solved the missions with and without the replanning aspect, and each of their runtimes are stated in table 5.10. Furthermore, the advantage of implementing AI Planning methods are

The fastest method for both the missions with and without replanning was HTN. For the mission planning problem without the replanning aspect, i.e. when the correct tool is already installed, the HTN method was 11 times faster than Graphplan in finding a solution, while for the mission planning problem with the replanning aspect the HTN method was 13 times faster.

Additionally, the HTN method was twice as fast at solving the mission planning problem without the replanning aspect compared to the mission planning problem with the replanning aspect. Furthermore, the first resulted in a solution at depth 39 while the latter reached a solution at depth 43. The higher level of depth justifies the additional time used for solving the mission planning problem with the replanning aspect.

	Correct Tool Installed	**Incorrect Tool Installed**
HTN	0.007s	0.013s
Graphplan	0.0625s	0.0951s

Table 5.10: Runtime of the implemented AI Planning methods

The mission planning problem defined for the Graphplan algorithm was to some extent simplified, as it was divided into three sub-missions. By having a reduced state-space, it is easier for the algorithm to converge to a solution as the planning graph which is created by the Graphplan algorithm is smaller. Furthermore, a graph searching algorithm will converge faster with a smaller planning graph and consequently the overall runtime would be better.

Another point worth mentioning is how the defined mission planning problem is divided for both the Graphplan and Q-learning algorithm, as the division might affect how the algorithms would have solved the problem. The reason for dividing the mission planning problem was to increase the chances of the algorithms to converge towards a solution. However, one cannot ensure that the overall solution which is obtained by combining the solution for each sub-mission is identical to the solution the algorithms could potentially obtain by solving the mission as only one. It might be necessary to model the mission planning problem differently, to reach a solution for the whole mission as one. Conse-

quently, this would potentially also increase the runtime.

It is worth mentioning that the defined mission planning problem is a simplified version of a potential IMR mission. Therefore, the variation in the different runtimes described in table 5.10 are in terms of $1/100$ of a second. By increasing the complexity of the mission planning problem, one might see bigger differences in the overall runtime as wel.

Chapter 6

Conclusion

This thesis has carried out the work of defining a mission planning problem which reflects IMR operations in the subsea domain, altogether with the feedback gained from the industry. The defined mission was The mission planning problem was quite a simplification compared to a real IMR operation, but did still result in valuable insight about the implemented methods; Graphplan, HTN and Q-learning, in addition to their corresponding strengths and weaknesses. This sets the ground for further development as it gives an indication of which method could potentially match with some problem definition.

The mission planning problem was defined in such a manner that it required minimal modification for each of the implemented methods. Furthermore, it was modeled in PDDL which is the de facto standard for modeling planning problems within AI Planning. The mission planning problem was implemented similarly for the Graphplan algorithm and the Q-learning algorithm, as both of them requires a state-space representation.

In regards to the implemented methods, the easiest comparison is between the state-space planner Graphplan and the plan-space planner HTN as these methods solved nearly the same two missions. It is worth mentioning that the latter had a better runtime for both the missions, in addition to solving the mission as a whole. The division into sub-tasks is how the the method works, and this is therefore not considered as a design modification. On the contrary, the defined mission planning problem was modified for the Graphplan algorithm. First and foremost, the mission planning problem was divided into three submission where the algorithm solved each of them subsequently. Upon completion of the three sub-missions, the solutions from each of the respective missions were accumulated to an overall solution. The drawback of finding an overall solution in this manner is the fact that one cannot guarantee that the method would have solved the problem similarly, and consequently the overall solution could potentially be wrong.

The defined mission planning problem is simplified when solved by the Q-learning method. This is due to the fact that a simple reward function was implemented, which did not have the capability of revisiting the same states before reaching the goal state. For instance if the initial state was the docking station and the the goal state was also the docking station, then the defined reward function would not be able to guide the agent into inspecting and operating the valves. The agent will consider to have completed the mission planning problem, as the initial state and goal state coincides. Since the defined mission for Q-learning was a simplification of the overall mission planning problem, it is not possible to compare it directly with the AI Planning methods. However, it is illustrated in this thesis that a RL method such as Q-learning has the potential of solving a mission planning problem.

Chapter 7

Further Work

In terms of further work, there are many possibilities. First and foremost, the modeling of the IMR mission could be modified such that the domain includes more details. This would increase the complexity of the mission, and consequently be more realistic compared to the actual IMR operation. Furthermore, the level of uncertainty could also be increased by having multiple cases for which the agent has to replan how to achieve the predefined goal.

Furthermore, the Q-learning algorithm could be improved by implementing a reward function which is able to represent all the aspects in the overall mission planning, and consequently there would be no need to divide the overall mission into sub-missions. This would mean that the agent would take into consideration which states it has already explored and which states require the agent to re-visit in order to complete the defined mission planning problem.

Another aspect, which would be useful in the development of Reinforcement Learning methods for solving mission planning problem is a simulator. The custom environment implemented in this master's project is compatible with the OpenAI Gym, which has the ability to connect to pre-existing simulators. It is also possible to implement a simulator for the defined domain of IMR operation with the corresponding installations such as a docking station, a warehouse and a subsea panel.

An additional possibility is to use Deep Reinforcement Learning instead of Q-learning, if the state-space of the mission planning problem increases. If the state-space is too large, then regular RL methods such as Q-learning struggle to find a final solution.

Lastly, the initial topic for this master's thesis, which was the connection between the EUROPA planner and the low-level control architecture T-REX. The incorporation of a planner such as EUROPA together with a low-level controller could also be part of the future work within AI planning for IMR operations.

Bibliography

[1] Sarah Sayeed Qureshi. Mission planning for underwater intervention drones by using automated planning and scheduling. unpublished, 2019.

[2] Libo Xue and Anastasios M. Lekkas. Ai planning for underwater intervention drone. *Oceans*, 2020.

[3] Dana Nau. Pyhop version 1.2.2. Available at `https://bitbucket.org/dananau/pyhop/src/default/`.

[4] Stuart Russel and Peter Norvig. Artificial intelligence: A modern approach - online code repository. Available at `https://github.com/aimacode`.

[5] Greg Brockman and Jie Tang. Openai gym. Available at `https://github.com/openai/gym`.

[6] Ingrid Schjølberg and Ingrid Bouwer Utne. Towards autonomy in rov operations. *IFAC-PapersOnLine*, 48(2):183–188, 2015.

[7] Narcís Palomeras, Arnau Carrera, Natàlia Hurtós, George C Karras, Charalampos P Bechlioulis, Michael Cashmore, Daniele Magazzeni, Derek Long, Maria Fox, Kostas J Kyriakopoulos, et al. Toward persistent autonomous intervention in a subsea panel. *Autonomous Robots*, 40(7):1279–1306, 2016.

[8] David M Lane, Francesco Maurelli, Petar Kormushev, Marc Carreras, Maria Fox, and Konstantinos Kyriakopoulos. Persistent autonomy: the challenges of the pandora project. *IFAC Proceedings Volumes*, 45(27):268–273, 2012.

[9] Steve Chien, Ari Jonsson, and Russell Knight. Automated planning & scheduling for space mission operations. 2005.

[10] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[11] M. Carreras, P. Ridao, R. Garcia, and T. Nicosevici. Vision-based localization of an underwater robot in a structured environment. In *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, volume 1, pages 971–976 vol.1, Sep. 2003.

[12] M. Prats, J. C. García, S. Wirth, D. Ribas, P. J. Sanz, P. Ridao, N. Gracias, and G. Oliver. Multipurpose autonomous underwater intervention: A systems integration perspective. In *2012 20th Mediterranean Conference on Control Automation (MED)*, pages 1379–1384, July 2012.

[13] M. Carreras, J. Batlle, and P. Ridao. Hybrid coordination of reinforcement learning-based behaviors for auv control. In *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180)*, volume 3, pages 1410–1415 vol.3, Oct 2001.

[14] K. B. Enonsen and O. K. Hagen. Recent developments in the hugin auv terrain navigation system. In *OCEANS'11 MTS/IEEE KONA*, pages 1–7, Sep. 2011.

[15] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen. A deliberative architecture for auv control. In *2008 IEEE International Conference on Robotics and Automation*, pages 1049–1054, 2008.

[16] Ingrid Schjølberg, Tor B Gjersvik, Aksel A Transeth, and Ingrid B Utne. Next generation subsea inspection, maintenance and repair operations. *IFAC-PapersOnLine*, 49(23):434–439, 2016.

[17] David M Lane, Francesco Maurelli, Petar Kormushev, Marc Carreras, Maria Fox, and Konstantinos Kyriakopoulos. Pandora-persistent autonomy through learning, adaptation, observation and replanning. *IFAC-PapersOnLine*, 48(2):238–243, 2015.

[18] Oceaneering. Freedom^{TM} resident rov. Available at `https://www.oceaneering.com/brochures/freedom-rov/`.

[19] "Eelume Subsea Intervention". The eelume concept. Available at `https://eelume.com/#the-story`.

[20] Equinor. Here are six of the coolest offshore robots. Available at `https://www.equinor.com/en/magazine/here-are-six-of-the-coolest-offshore-robots.html`.

[21] Tormod Haugstad. Den kan gå minst 20 mil på batteri og jobbe på 6.000 meters dyp i 6 måneder. Available at `https://www.tu.no/artikler/den-kan-ga-minst-20-mil-pa-batteri-og-jobbe-pa-6-000-meters-\dyp-i-6-maneder/473674`.

[22] Conor McGann, Frederic Py, Kanna Rajan, John P Ryan, and Richard Henthorn. Adaptive control for autonomous underwater vehicles. In *AAAI*, pages 1319–1324, 2008.

[23] Steve Chien, Benjamin Smith, Gregg Rabideau, Nicola Muscettola, and Kanna Rajan. Automated planning and scheduling for goal-based autonomous spacecraft. *IEEE Intelligent Systems and their applications*, 13(5):50–55, 1998.

[24] Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sebastien Racanière, David Reichert, Théophane Weber, Daan Wierstra, and Peter Battaglia. Learning model-based planning from scratch. *arXiv preprint arXiv:1707.06170*, 2017.

[25] Sergio Jiménez, Tomás De la Rosa, Susana Fernández, Fernando Fernández, and Daniel Borrajo. A review of machine learning for automated planning. *The Knowledge Engineering Review*, 27(4):433–467, 2012.

[26] Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, 2017.

[27] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[28] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[29] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

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

[31] Kenneth H Rosen. *Handbook of discrete and combinatorial mathematics*. CRC press, 1999.

[32] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[33] Kutluhan Erol, James Hendler, and Dana S Nau. Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994.

[34] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *Journal of artificial intelligence research*, 20:379–404, 2003.

[35] Dana Nau, T-C Au, Okhtay Ilghami, Ugur Kuter, Dan Wu, Fusun Yaman, Héctor Munoz-Avila, and J William Murdock. Applications of shop and shop2. *IEEE Intelligent Systems*, 20(2):34–41, 2005.

[36] Dana Nau, Héctor Munoz-Avila, Yue Cao, Amnon Lotem, and Steven Mitchell. Total-order planning with partially ordered subtasks. In *IJCAI*, volume 1, pages 425–430, 2001.

[37] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[38] Stephen Marsland. *Machine learning: an algorithmic perspective*. CRC press, 2015.

[39] Ian Richter. Supervised vs. unsupervised machine learning. Available at https://blog.seebo.com/supervised-vs-unsupervised-machine-learning/.

[40] Olivier Sigaud and Olivier Buffet. *Markov decision processes in artificial intelligence*. John Wiley & Sons, 2013.

[41] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[42] A. Bondu, V. Lemaire, and M. Boullé. Exploration vs. exploitation in active learning : A bayesian approach. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2010.

[43] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[44] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language, 1998.

[45] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[46] Yong Bai and Qiang Bai. *Subsea pipelines and risers*. Elsevier, 2005.

[47] Envirent. Manipulator torque multiplier tool. Available at https://envirent.no/products/rov-tooling/torque-tool/manipulator-torque-multiplier-tool/.

[48] Envirent. Multipurpose cleaning tool round brush. Available at https://envirent.no/products/rov-tooling/cleaning-tool/multipurpose-cleaning-tool-round-brush/.

Sarah Sayeed Qureshi

NTNU
Norwegian University of
Science and Technology