NTNU
Norwegian University of
Science and Technology

Automated Planning and Control for a Simulated Robot

Magnus Aarskog

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Anastasios Lekkas, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

The early work in robotics and Artificial Intelligence showed great promise, but because of the challenges and difficulties met in the early phases, the two fields drifted apart. Artificial Intelligence focused more on algorithms and the abstract method of approaching problems, while the robotics aspect focused more on electrical and mechanical engineering. Now with the recent developments in Machine Learning, big data, computing power, sensors, software etc., the two paths these to fields have been on are getting closer [1].

The goal of this thesis is to try to combine Artificial Intelligence and robotics. The author has the most experience with robotics, and will therefore try to focus on the Artificial Intelligence part by making a fully usable planner. A planner, in this case, means a program that will find a sequence of actions that will lead to the desired goal. All the code for the planner has been written from scratch including a parser that will read the problem description files which the planner will utilize to find a solution to the planning problem.

To test the planner on a robotics system, the robot named KUKA YouBot is used to solve different planning problems such as Tower of Hanoi, a stack/restack problem of blocks and moving around in a domain where the robot must interact with the environment to complete its goal.

As mentioned above, the AI community has focused much on algorithms and the

abstract thinking around it. The problems that the planning algorithms have been based on have been in a deterministic matter where everything is known, which is not a realistic assumption of the real world where uncertainty plays a big part. The planner is based on a deterministic model where everything is known. This thesis will therefore make an attempt to adapt the planner such that it also can handle cases where not everything is known from before.

Sammendrag

Kunsitg intelligens og robotikk viste i sine tidlige stadier lovende resultater, men på grunn av utfordringene og vanskelighetene som opptsto i de tidlige fasene drev disse feltene fra hverandre. Kunstig intelligens fokuserte mer på algoritmer og de abstrakte metodene ved å løse problemer, mens robotikken fokuserte mer på elektronikk og det mekaniske aspektet. Nå som det har blitt gjort store fremskritt i maskinlæring, big data, regnekraft, sensorer, programvare etc., begynner nå robotikk og kunstig intelligens igjen å nærme seg hverandre [1].

Målet med denne oppgaven er å forsøke å kombinere kunstig intelligene og robotikk. Forfatteren har tidligere erfaring innenfor robotikk og vil derfor fokusere på kunstig intelligens-delen ved å lage en fullt brukbar planlegger. En planlegger betyr i denne sammenhengen et program som vil finne en sekvens med handlinger/ordre som vil lede til et ønsket mål. All koden for planleggeren har blitt skrevet fra bunnen av, inkludert en parser som vil tolke filene som beskriver problemet som planleggeren skal løse.

For å teste planleggeren på et robotsystem vil roboten kalt KUKA YouBot bli brukt til å løse forskjellige planleggingsproblemer. Problemer som Tårnet i Hanoi, et stablings/omstablingsproblem av klosser og å bevege seg rundt i et miljø hvor roboten må interagere med omgivelsene for å nå målet sitt.

Som nevnt over, så har kunstig intelligens foksuert mye på algoritmer og det abstrakte

rundt dette. Plannleggingsproblemene som algoritmene har blitt basert på har vært deterministiske hvor alt er kjent på forhånd. Dette er ikke en realistisk antagelse av den ekte verden hvor usikkerhet spiller en stor rolle. Planleggeren er basert på en deterministisk model hvor alt er kjent. Derfor vil denne oppgaven også fokusere på å tilpasse planleggeren slik at den også kan fungere hvor alt ikke er kjent.

Preface

This thesis was written during Spring 2018 for the Department of Engineering Cybernetics at Norwegian University of Science and Technology(NTNU).

Thank you to my supervisor Anastasios Lekkas for guiding me through this thesis. Also thanks to my family and friends for moral support.

In this thesis the following tools have been used:

- The Linux distribution Ubuntu 16.04
- Python 2.7 programming language [2]
- V-REP simulation software [3]
- Robot Operating System (ROS) [4]

All the results from the planner that have been generated by the code that I have written. No additional code has been given to me. Only the basic libraries from Python 2.7 have been used.

In V-REP and for the KUKA YouBot there is an example code that is given which is used as a starting point for further development, i.e. the controller and kinematics for the robot is not made by me, but the framework for communication between the planner and the robot using ROS is made by me.

Problem description

The tasks in this thesis are to:

- Create a planner that can solve planning tasks described by an action language.
- Test the planner on a real-time system in a simulation environment.
- Explore the possibilities of adding uncertainty to the simulation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Previous Work

When Artificial Intelligence (AI) and Robotics where in their early phases, these two fields were strongly connected. Projects such as Shakey the robot [5] in 1966-1972 had a big impact on both AI and robotics. The results led to the development of the A* search algorithm [6], which will be further explored later in this thesis. Also, the Hough Transform [7], which is a method used in image processing for feature extraction, was improved further because of this [8]. But because of the challenges of making a complete intelligent system, the fields diverged. This has led researchers only to focus and specializing on specific topics separately, leading to little cooperation between these two fields. There are now several advancements in the recent years that makes the AI and robotics field ready to be further developed. Advancements such as computing power, machine learning and big data for instance. This has led the science of robotics and Artificial Intelligence to become more prevalent. Over the recent twenty years, the robotics field has been greatly improved in terms of sensors, control algorithms, electrical engineering etc. The number of open source platforms

is increasing leading to even more development in the field. Because of this, and the advances in navigation, manipulation and perception, robotics is expected to be one of the fastest growing markets in the next years [9].

Since the robots are now starting to make big advances into markets such as hospitals, service related workplaces and production, new demands towards versatility and the ability to learn, plan, adapt and interact will arise. These topics have been the focus of AI the last decades and therefore the two paths robotics and AI has been on over the last years seems to get closer to each other. It is believed that the integration of AI and robotics will lead to much disruptive innovation in the time to come [1]. The editorial [1] also explains that the typical curriculum for a student is either robotics or AI, but rarely both. In this thesis, a student with knowledge about robotics will try to implement an AI to a simulated robotic system so as to become aware of the challenges involved when attempting to close the gap.

As the two fields now are reuniting, the investments in commercializing AI technologies to robotics is increasing. For example, the Bossa Nova robot [10] which is able to react and plan in a dynamic environment without human intervention. Earlier projects such as Flakey [11] which is the successor to Shakey and the museum tour-guide robot [12] has also been contributing to the combined AI and robotics field. More interesting is maybe the Deep Space 1(DS1) [13, 14] project from NASA where its mission was to fly by a comet and an asteroid. It was also meant to test out new technology. DS1 was the first spacecraft which operated without human intervention.

The goal of this thesis is therefore to create a fully usable AI planner which can solve basic planning problems. Although open source planners already exist [15, 16], the author has no former experience with this topic and making a planner from scratch will give good insight into the world of AI planning.

Autonomous vehicles are more and more relevant in these days. Self-driving cars and autonomous boats are heavily invested in. With this, it is expected that the autonomous entities are taking the best and most optimal decisions based on how the world is

perceived.

In this thesis, an approach to use the self-made planner to make a plan for an autonomous robot, which has a manipulator to aid its work, towards a goal in a world where some level of uncertainty is present. This is made to try to create an autonomous vehicle that needs as little human intervention as possible.

Since this is a rather large area for one person to fully cover for one semester, this thesis is focusing on making a framework or create awareness for further research into AI planning and replanning for autonomous systems.

1.2 Outline of Report

This report can be divided into two main parts. The first part which includes Chapter 2, 3 and 4 where Chapter 2 gives some basic knowledge about planning and how it is defined in this thesis. Chapter 3 includes the making of a planner which parses from files describing the world and the problem to be solved into a usable format for the solver which is described in Chapter 4.

The second part of the thesis which is described in Chapter 5 covers some use-cases of planning where some level of uncertainty is present and an implementation of a planning system for the KUKA YouBot where the robot is to solve different tasks using the planner made in the first part of the thesis.

Chapter 2

Background

This chapter will go through some basics of planning and give a description of what planning actually is.

2.1 Automated Planning

Planning is an explicit deliberation process where the actions are chosen and organized by anticipating the expected outcomes of each action. An *action* is used in the setting of something that an agent does. This can be things like doing a motion, communication, perceive something or a force which will make a change to the environment or state of the agent. The agent must be able to interact with the environment. The agent wants to complete one or more objectives using actions. To achieve the objective, a deliberation process deciding which actions to execute in order to achieve the objective is performed [17].

Deliberation consists of choosing which actions to do and how they are used to complete an objective. The reasoning process includes what the result of the action is

and what actions to undertake in order to get the desired effect. This process gives the agent the ability to predict and decide how to combine actions such that they together complete the wanted effect. For example:

- When you are following a car driving down the road with your sight, you move your head, eyes, and body.
- When you are fishing you must first obtain a fishing rod, put on a bait and then throw the bait into the water to obtain a fish.
- To be able to make an origami bird from a sheet of paper, one have to do a series of actions where the combined effects of the actions will lead to the sheet of paper looking like a bird.

The first bullet point expresses the coordination and sensing of the body in order to track an object. The actions which aim at keeping the car in the field of view is purposeful but is more acting than deliberation. The last two scenarios are tasks where a set of reasoning must happen in order to complete the goal [17]. This shows the nature of acting deliberately.

On an everyday basis, our actions require some planning while others not so much. When we are acting, we anticipate the outcome of our actions even though we do not know the full outcome of the action. Normally, we are acting much more than we deliberate. Much of what we do are based on experience which makes us do complicated tasks without actually planning for them.

When humans must plan it is because the activity addresses a new situation and/or complex tasks which is not so familiar. People mainly do deliberation in cases where the risk is high or when the cost is high, i.e. we only plan when it is strictly necessary because the time consumption and cost can tend to be very high as well as complicated. When we deliberate we often come up with plans that are feasible and good instead of the most optimal plan.

In Artificial Intelligence(AI), automated planning studies the deliberation process computationally. Automated planning can be very beneficial. Where the systems tend

to be large and the resources and time constraints are important, automated planning can do a job much faster and better than a human operator. An example can be in space-related planning e.g. a Mars rover which has limited resources based on time and power and at the same time one wants to collect as many samples as possible and take pictures of points of interest. Another use can be in shipping and transport where a system can contain a lot of different ships, planes and other transport vehicles. Finding an optimal plan to transport the cargo to its desired locations can be a tedious and difficult task for a human [18].

The focus of the rest of the chapter will be to get an overview of how to represent planning domains such that a computer can read them and solve the planning problems that are defined.

2.2 Classical Planning

This section gives an overview of classical planning and is mostly based on [17]. The models that describe planning systems are called planning domains. The planning domain is an approximation of the agent and the environment that emphasizes on understandability. The agent and domain can be represented using a state-transition system [17]. In [17], the authors defines the state transition system as a triple $\sum = (S, A, \gamma)$ or if the actions has some costs related to them it becomes a 4-tuple: $\sum = (S, A, \gamma, \text{cost})$ where:

- S is a set of states the system can be in.
- A is a set of actions the agent can do in the system.
- γ is a function that keeps track of what actions that are applicable in a given state.
- *cost* is a function that represents the cost of an action in terms of money, time consumption etc.

The definition above is called a classical planning domain, where a set of assumptions must be satisfied in order for it to be usable:

- The environment must be static and finite. This means that the changes that happen are only because of actions being executed. This means that other agents or external events cannot intervene.
- There is no time dependency. All actions happen in sequence and are not based on when to start the action, or if the actions are performed concurrently.
- All the actions are deterministic. I.e. the effects of an action is always known.

This forms the basis of classical planning and the next subsections will focus on how to express a planning system based on classical planning.

Most of the action languages are based on first-order logic. From the first order logic there are predicates which are functions that take in an object or any entity and returns *true* or *false*. The states are often defined with *ground atoms*,which means that it does not contain any free variables, e.g. the atom *at(Heathrow,plane2)* where *at(x,y)* is the predicate function

2.3 Situation Calculus

The situation calculus was first described in 1963 by McCarthy [19]. The situation calculus is a logical language that represents changes in scenarios. The very basic concept of the situation calculus are the situations, actions and fluents [20].

- **Situations**
  Situations have multiple definitions. According to [20] a situation is the state of the world at a certain time instance. According to [21], a situation is the finite sequence of actions leading from the initial situation S_0 to the current situation and is the definition that is used.
- **Actions**

An action marks the transition between two situations of the world. An example is *move(location)* where the entity is to move to *location*. An action consists of a set of preconditions and effects. The preconditions state if the action is executable in a given situation. If an action is possible, the effects give the effect of the action on the fluents.

- **Fluents**
  Predicates with arguments that depend on time are called fluents. Fluents are used to describe the situations as well as the effects of actions. Fluents can have two forms: relational fluents and functional fluents. Relational fluents are described as *true* or *false* statements. An example of a relational fluent can be a fluent called *at_table(pencil)* which states if the object *pencil* lies on the table or not. The other type of fluents can take a range of different values. For example, *location* which returns the location of the pencil or some other kind of object [22].

2.3.1 Example

A classical example, which will also be used to test the planner later on, is the blocks world. The blocks world consists of a given number of blocks sitting on a table. The final goal is to rearrange the blocks into one or more stacks in a specific order. Only one block can be moved at a time and the blocks can either be moved onto the table or onto another block. If a block has another block on top of itself, it can not be moved. To describe a situation calculus domain one must find the actions available for the agent to perform and the fluents which will describe the state of the world and the effects the actions will have on the world. The actions of the blocks world are:

- *stack(x,y)* - this action puts block x onto block y given that the robot is holding block x and there is no block on top of block y, i.e. block y is clear.
- *unstack(x,y)* - the opposite action of *stack* which is pick up block x from block y given that block x is clear and the gripper is empty.

- *putdown(x)* - if the robot is holding block x then it is possible to put it down on the table.
- *pickup(x)* - if the robot is not already holding anything, the robot can pick block x from the table provided the block is clear.

This describes the four different actions that the robot can do. The fluents that can be used are:

- *holding(x)* - which is *true* if the robot is holding block x
- *on(x,y)* - this fluent is *true* if block x is on block y
- *ontable(x)* - this returns *true* if block x is on the table.
- *clear(x)* - this is *true* if block x does not have any blocks on top of itself
- *handempy* - true if the gripper does not hold any of the blocks.

If the robot are going to do the *stack(x,y)* action in a given situation. Then the fluents *holding(x)* and *clear(y)* must be true. After this action has been performed the effects will be that *on(x,y)* and *handempty* is *true*, and *holding(x)* and *clear(y)* are *false*.

2.4 STRIPS

The situation calculus is sometimes referred to as too descriptive, leading to that it may be harder to develop good solving algorithms. One particular problem arises when actions are executed and one must also specify the non-effects as well, i.e. what *does not* change when an action is executed. This is called the Frame Problem [20] and it also occurs when defining the state. Because of this, the state can get very large, which is not desirable. STanford Research Institute Problem Solver (STRIPS) [23] is one of the approaches which tries to solve some of the shortcomings of the situation calculus such as the Frame Problem and to be more restrictive in its declarations. The way STRIPS deals with the Frame Problem is to assume that an action only changes a feature of the world if it says so [24]. This means that if one is going to pick up a cup

from a table, one does not have to state that the table is still there after the cup has been picked up. STRIPS atoms that are not mentioned in the state are *false*. E.g. if the *handempty* atom is not present in the current state, it is *false*. This is called the Closed World Assumption(CWA). The *effects* property of a STRIPS action consists of the *delete list* and *add list*. The *delete list* states what atoms that are deleted from the state and the *add list* describes what atoms to add to the state after the action has been executed. For example, in the blocks world the *stack(x,y)* actions *delete list* will be *holding(x)* and *clear(y)*, and the *add list* will be *on(x,y)* and *handempty*.

To define a complete STRIPS instance one needs to define an initial state, a set of goal atoms that the planner wants to reach and a set of actions. For the blocks world a STRIPS instance can look something like this:

- Initial state: *on(A,B)*, *ontable(B)*, *ontable(C)*, *clear(A)*, *clear(C)*, *handempty*
- Goal: *on(C,A)*
- Actions:
  - *stack(x,y)*
    Preconditions: *holding(x), clear(y)*
    Add list: *handempty, on(x,y)*
    Delete list: *holding(x), clear(y)*
  - *unstack(x,y)*
    Preconditions: *handempty, on(x,y), clear(x)*
    Add list: *holding(x), clear(y)*
    Delete list: *handempty, on(x,y)*
  - putdown(x)
    Preconditions: *holding(x)*
    Add list: *ontable(x), handempty*
    Delete list: *holding(x)*
  - *pickup(x)*
    Preconditions: *clear(x), handempty, ontable(x)*

Add list: *holding(x)*
Delete list: *handempty, ontable(x)*

A possible solution for this problem is first *pickup(C)* and then *stack(C,A)*.

One of the shortcomings of STRIPS is the inability to express conditional effects of an action [25] i.e. the effect may state *when(condition,effect)*. The action language Action Description Language(ADL) [26] was made to solve this. Some of the differences between STRIPS and ADL are that STRIPS is based on a Closed World Assumption(CWA), and ADL is based on an Open World Assumption(OWA) where a statement can be *true* whether it is known or not. This means that in the CWA if the atom is not defined in the state it is *false* while in the OWA it would be *unknown*. Because of the simplicity of STRIPS and the CWA, the STRIPS structure will be the language that will be used in this thesis.

2.5 PDDL

The Planning Domain Definition Language [27] was made to make a standard of AI planning and make it easier to compare and test different systems and approaches which will lead to a faster progress in the field. PDDL is therefore used in this thesis to express the STRIPS problems.

The whole PDDL system is given in two files. The first file is a domain file normally named *domain.pddl*. The domain file states what available predicates can be used to declare the planning domain. It also gives all the actions the agent is able to do. The second PDDL file that is used, describes the problem at hand and is mostly called, or something similar to, *problem.pddl*. This file gives the initial state the domain is in and the goal atoms that need to be satisfied.

2.5.1 PDDL Example

To get a feel of how the PDDL files are set up, and a more descriptive way of how to define a STRIPS problem in PDDL, consider a simple example where two objects are swapped between two locations. The first part of the domain file is given in Code 2.1. The first line in Code 2.1 only defines the domain name. The second line states what requirements the planner must support in order to solve problems defined in this domain. In this case, the STRIPS requirements will be considered. The predicates field declares the predicates for the domain. This means that this is the objects which will be used to define the domain and state [27].

```
(define (domain swap)
  (:requirements :strips)
  (:predicates
    (At ?obj ?loc)
    (object ?obj)
    (location ?loc) )
    ...
```

Code 2.1: PDDL domain code example

In Code 2.2 the rest of the domain file is seen. Here, the only action the agent can do is defined. The name is *swap_objects* and it takes in four parameters. To able to execute the action, the current state of the domain must satisfy the preconditions of the action. In this case *obj1* and *obj2* must be objects and *loc1* and *loc2* must be locations. The objects need to be at the two different locations to be able to execute the action. When the action has been executed, one needs to consider the effects of the action. In Code 2.2 the *:effect* tab declares what states that is added to the current state and what states to delete from the current state i.e. *add list* and *delete list*. The *not* part declares that this atom is to be added to the *delete list*. The *and* statements that one can see in the code states that all the stated atoms must be true. In this case, since STRIPS only allows conjunctions, it is redundant. But, if one wishes to use ADL, which allows disjunctions (*or*) it is relevant.

```
...
  (:action swap_objects
```

```
    :parameters (?obj1 ?obj2 ?loc1 ?loc2)
    :precondition (and
       (At ?obj1 ?loc1) (At ?obj2 ?loc2) (object ?obj1)
       (object ?obj2) (location ?loc1) (location ?loc2))
    :effect (and (
       (At ?obj1 ?loc2) (At ?obj2 ?loc1)
       (not (At ?obj1 ?loc1)) (not (At ?obj2 ?loc2))
       ))))
```

Code 2.2: PDDL domain code example

Code 2.3 defines a problem in the domain created in Code 2.1 and Code 2.2. The first line states the name of the problem. The next line states what domain the problem is defined in. In *:objects* the objects for this problem is defined. In this example, there are two boxes, a floor and a table. The *:init* section defines the initial state of the domain. From the initial state one can see that the boxes are defined as objects and *floor* and *table* as locations. *Box1* is at *floor* and *box2* is at *table*. The last element is the *:goal*. This contains the atoms that need to be fulfilled for the problem to be solved.

```
(define (problem problem0)
    (:domain swap)
    (:objects box1 box2 floor table)
    (:init (object box1) (object box2) (location floor)
    (location table) (At box1 floor) (At box2 table))
    (:goal (and (At box1 table) (At box2 floor)))
)
```

Code 2.3: PDDL problem code example

This example shows how the PDDL language is built up. The domain and problem are split up into two files. One as a domain file and one as a problem file. The solution in this example will simply be to do the action *swap(box1,box2,floor,table)* where the state will satisfy the goal conditions. In most of the other cases, the solution will be a sequence of action which, if possible, will lead to a state where the goal is satisfied.

Chapter 3

Implementation - Parser

Now that some background on STRIPS and PDDL has been covered it is time to implement the first part of the planner. The parser will parse the domain file and the problem file such that the solver can get the applicable actions in a given state to look for a solution. In Figure 3.1 one can see an overall plan of how the process of finding a plan is imagined to be. The figure shows that the two files are parsed by the parser then the solver uses the generated data to find a sequence of actions that leads to a state where all the subgoals are satisfied.

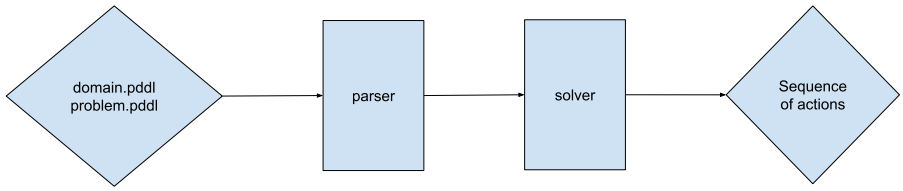


Figure 3.1: Plan

Initially, there will be two main classes that are used to read the files. The first, which

reads the domain file, will extract all the information from the file into this class. The class that reads the problem file is used to define the initial state. This class is used to represent each new state that is found from the initial state.

3.1 Domain Parser

As mentioned above, the domain class contains all the actions of the domain. Each action is declared as a class itself. In Code 3.1 the declaration of the domain class is given. As one can see, the init function takes the domain file as input and passes it on to the parser function.

```
class Domain:

    def __init__(self,domain_file):
        self.domain_name = ''
        self.requirements = []
        self.predicates = []
        self.actions = []
        self.parse(domain_file)
```

Code 3.1: Python domain class

The way the parser works is to count the parentheses in the file. The code iterates through the file until it finds a parenthesis and if it is a left parenthesis, '(', it adds one to the counter and the right parentheses, ')', subtracts one. It will be possible to extract the different sections of the file by looking at when the counter is zero. This function is used several times to also extract the individual predicates, parameters, preconditions and so on throughout the code.

In Code 3.2 one can see the function *set_domain_property()* which takes in the different sections from the code. The elements that are sent in can look like *(:predicates(At ?obj ?loc)(object ?obj)(location ?loc))* and then if-else statements updates the domain according to what kind of section that is the current extract.

```
def set_domain_property(self,element):
    element = "".join(element)
    element = element.lower()

    if element[1:7]=='define':
        self.domain_name = element[14:-1]

    elif element[1:14]==':requirements':
        self.requirements = element[15:21]
        if self.requirements!='strips':
            raise ValueError('Error: ',element,"Wrong requirements, must be
                STRIPS")

    elif element[1:12]==':predicates':
        self.set_predicates(element[12:-1])

    elif element[1:8]==':action':
        self.actions.append(Action(element[8:-1]))
    else:
        raise ValueError('Error: ',element,'Cannot recognize this property.
            ')
```

Code 3.2: Python define domain

The predicates list in the domain class is a list of predicate classes. In Code 3.3 the predicate class only states its name and parameters.

```
def __init__(self,predicate):
    self.parameters=[]
    self.name=''
```

Code 3.3: Python predicate class

The action class is built up in the same way as the domain class. Instead of using the whole domain file to define the domain, the action class *init* function takes in each string containing the action statement and parses this. In Code 3.4 one can see the function is similar to Code 3.2.

```
if element[0]!=':':
    self.set_name_and_parameters(element)
elif element[0:13] == ':precondition':
    self.set_preconditions(element)

elif element[0:7]==':effect':
    self.sef_effects(element)
else:
       raise ValueError('Error in: ',element,"Did not recognize action
           property")
```

Code 3.4: Python code creating the action class

The *set_preconditions()* fucntion seen in Code 3.4 extracts the different preconditions and add them in as classes in a precondition list as a action class property. The *set_effects()* function gets the effects and add them to an *add_list* variable for the effects and a *delete_effects* for the delete effects. This means that the *add_list* list will be added to the state when the actions is executed, and the *delete_effects* will be deleted from the state when the action is executed.

3.2 Problem Parser

The second part of the parser is the problem parser. The problem parsers main class is the state class. This is the most important class in the parser. The state class is used to parse the file to create an initial state class and then the initial class is used in the solver to create a new set of states and so on until the goal state is reached via a set sequence of actions. The domain file and problem file are built up in the same way, so much of the code that was used in the domain parser can be used in the problem parser as well.

An important feature of the class is to find all the actions that are possible to do based on the current state. The function is called *create_child_states()* and it iterates through all the actions in the domain and sends in the current state to check if the

action can be executed based on the state. The first solution was to try all combinations of the state objects as input to the function. This means that one can use a dynamic number of nested for loops based on how many inputs the action has. This can be solved using recursion. This solution does not scale well and if the number of inputs gets large, and the number of objects is large, this solution will use a long time to find all the possible actions. This is a brute force kind of method to solve this problem.

Another method of doing this is more involved, but also a lot faster than the first solution. The way this method works is to first extract the relevant atoms for each of the actions. The next step is then to map the objects from the atoms to the input of the action, e.g. if the action *unstack(x,y)* from the example in Section 2.4 is considered. The relevant atoms in the goal state are *on(A,B)*, *on(C,A)*, *handempty* and *clear(C)*. The algorithm will check if *handempty* is *true*, which in this case it is. The next step is to use the input of the action *(x,y)* and map them to the preconditions. The first atom is *on(A,B)* which corresponds to the precondition *on(x,y)*. This means that *x=A* and *y=B* will be the assignments. The next precondition is *clear(x)* which corresponds in this case to the atom *clear(A)* since this atom does not exist in the state, the action *unstack(A,B)* is not possible. The algorithm continues to the next atom which is *on(C,A)* and maps the objects accordingly. Here the atom *clear(C)* exist and all preconditions are satisfied, which means that *unstack(C,A)* is a possible action and will be added to the list of possible actions.

Since actions have a varying number of preconditions, this problem is solved with recursion. If all of the inputs are not mapped on the first precondition, it will be mapped in one of the next steps in the recursion.

Chapter 4

Implementation - Solver

In this chapter a solver to solve the STRIPS problem written in PDDL and parsed by the parser in Chapter 3 is made. It will be able to choose from different search algorithms which will traverse the state space until a state which satisfies the goals has been found, or return *fail* if a solution can not be found. A solution is not found when the solver has visited all the states where none satisfies all the subgoals.

4.1 Background

There are a lot of different approaches, algorithms and methods for solving a planning problem. This section will be a brief overview of how automatic planning is done in this thesis.

4.1.1 Forward Planning

In this project, the forward state space search is the strategy that will be considered. The forward state space search starts at the initial node and traverses the state space

until a goal node is found. The nodes here represent the individual states and the edges of the graph represent the actions [28]. Another possible solution approach is to use regression planning (backward search) where the start node is the goal node. In forward search, the nodes are expanded by looking at the applicable actions, which are available in the current state. In backward planning, the nodes are expanded by looking at the relevant actions, i.e. what actions are contributing to satisfy the goal state. If an action satisfies a subgoal it is seen as relevant. If an action also negates an element of the goal it is not seen as relevant since then you need at least one more step to complete the goal. Backward planning is not chosen because it is harder to come up with good heuristic functions [28]. What a heuristic function is, will be covered in the next section.

4.1.2 Heuristic Function

A heuristic function is a function that estimates the distance from a given node to the goal node. For example, when traveling from A to B via different nodes and edges e.g. cities and roads, the estimate of how long distance you have left is the straight line between you and B. In planning, a heuristic function is used to estimate how many actions that are left until a solution is reached. If a heuristic function is said to be admissible it will never overestimate the distance left. This means that it is possible to find an optimal solution using this heuristic function. If a heuristic is informative it will give a good estimate of the distance that is left to the solution. Therefore a heuristic function that is both admissible and informative is desirable but can be hard to find.

4.1.2.1 Missing Number of subgoals

One possible solution for a heuristic function is to count the number of missing subgoals. This method is not admissible, because one action can satisfy two subgoals and can therefore overestimate the number of actions left. Its informativeness will depend on the stated goal. If only one goal atom is stated, the heuristic function will

return 1 until a solution is found, which is not very good. But if more subgoals are stated, it can be more informative. By looking at the theory behind this heuristic, the performance using it will most likely vary.

4.1.2.2 Relaxed Problem Heuristic

A good heuristic function $h(s)$, where s is the state, is a heuristic function that solves the problem and returns the number of steps needed to complete the goal. Since this solution is pointless, because this actually solves the problem, one can consider a relaxed problem where the problem, is relaxed into a simpler problem which is easier to solve and then the length of this solution can be used as an estimate of the length to the original solution.

In [28, 29], one type of relaxed problem is defined as ignoring the *delete lists*. This means that only the *add list* is taken into consideration when trying to solve the relaxed problem. The state is therefore strictly expanding until a solution is found, or the state does not expand anymore. Unfortunately, the relaxed problem is still NP-hard [29]. This means that one needs an estimate of the optimal values of $h(s)$. This can be done by considering that all the applicable actions in that state are executed at each step and the positive effects are added to the relaxed problem state. This continues until the state satisfies all the sub-goals, or as mentioned above, the state stops expanding. Every time an action that satisfies one of the atoms in the goal, the cost, $g_s(p)$, which is the cost of achieving an atom p in the state s, is updated according to how many steps it took to achieve the sub-goal.

The heuristic function can be written as:

$$h(s) = \sum_{p \in G} g_s(p) \tag{4.1}$$

where G is the subgoals. The heuristic is informative, which is a good trait when considering this heuristic function approach. Another heuristic that can be extracted directly from the same method, is to instead of summing the cost of all of the actions

one can choose the subgoal with the largest cost. This heuristic is admissible, but not especially informative and will not be considered further.

Other relaxed problem heuristics also exist. It is possible to ignore preconditions such that all actions become applicable. Then count all the actions that together fulfill all the sub-goals. This is almost the same as counting all the missing subgoals as in the section above, but since one action can make more than one sub-goal to become *true*, it is not. Another relaxed problem heuristic is to ignore some specific predicates to make the state smaller and then solve the relaxed problem to get an estimate of the number of actions left [28].

4.1.3 Search Strategies

There exist many different graph traversal algorithms. The choice of a search strategy will affect how the solver traverses the graph to get to the solution and can greatly affect the time it will take to get to the desired node. The nodes are usually stored in a queue and the choice of solving algorithms decides how the nodes are arranged in the queue.

4.1.3.1 Breadth and Depth-First Search (BFS & DFS)

The breadth and depth-first search are the simplest and easiest search algorithms to implement. The BFS uses a *last in first out* (*LIFO*) queue. Which means that the oldest node that is discovered is the node to be explored next. The DFS uses a *first in first out* (*FIFO*) queue. This implies that the node that was discovered first is the node that is next to be explored.

4.1.3.2 Greedy Best-First Search

Greedy best-first search or also called greedy algorithm is a search technique which uses a heuristic function to determine what node that should be explored next. The

queue is sorted such that the node with the lowest heuristic estimate on the distance to the goal is chosen. This can lead quickly to a solution, but the length of the solution can be long.

4.1.3.3 A* (A Star)

The A* algorithm [6], is an algorithm used in pathfinding and graph traversal. It resembles the greedy best-first algorithm with one extra feature taken into account when selecting the next node. The A^* algorithm uses the distance from the initial node to the current node in its selection of node as well. The cost of the state is now the distance from the initial node, $g(s)$, added with the estimate of the remaining distance, $h(s)$, to the goal node. (4.2) shows the function calculating the state cost.

$$f(s) = h(s) + g(s) \tag{4.2}$$

The state, s, with the lowest value $f(s)$ is chosen to be the next node to be explored. When $h(s) = 0$ in (4.2), the algorithm is actually the Dijkstra's algorithm [30, 31], but since the cost for each action is one, it is also breadth-first search. For greedy best-first search, $g(s) = 0$, i.e. $f(s) = h(s)$.

If the heuristic function is admissible the A* algorithm is guaranteed to find the optimal solution [6]. Although sometimes it is more important to find the solution quick than to find the solution with the smallest number of steps one can decide to use Weighted A* search (WA*) instead [32], which is given by the function

$$f(s) = w \cdot h(s) + g(s) \tag{4.3}$$

This method may lead to solution of lower quality [33], but the speed will increase as well. If the heuristic function is admissible, the solution will not exceed the optimal solution by a factor of W. If $W = 1$, then (4.3) is the A* algorithm (4.2).

4.2 Implementation

The datasets that are generated in the process of traversing the domain can get large and it is important to implement the algorithm such that it uses the least amount of time looking through lists of states etc. There are mainly two lists that are used in this algorithm. As seen in the previous section 4.1.3 the best-first search chooses the node with the lowest state cost. One of the lists, in this case, must keep track of the visited nodes such that it will not visit the same node twice and thus also avoiding cycles. For this, it is natural to use a dictionary or a hash table which have an average look-up and insertion time complexity of $O(1)$. The key to the dictionary is the state. The algorithm will therefore check if the entry exists, and if not, the node is added as a new state.

The other list contains the nodes that have not been explored yet. The list is sorted based on the individual cost of each node. A good method to use here is to use a heap which has an average insertion time of $O(log(n))$ where n is the size of the heap. Since the lowest value is on the top of the heap, it takes $O(1)$ to get the lowest value. In Code 4.1 one can see a simple Python implementation of the suggested solver.

```
from heapq import heappush
from heapq import heappop

def solve(initial_node):

    heap = []
    visited_nodes = []

    heappush(heap,initial_node)

    '''While heap is not empty'''
    while heap:

        '''Get the node with the lowest cost from the heap'''
        current_node = heappop(heap)

        if current_node.is_goal():
            return current_node
```

```
        else:
            new_nodes = current_node.get_child_nodes()

            for new_node in new_nodes:

                if new_node not in visited_nodes:
                    new_node.set_node_cost()

                    '''Add the new node to visited nodes'''
                    visited_nodes[new_node.data] = True

                    '''Add the new state to the queue using a heap
                    sorted list based on the node cost:'''
                    heappush(heap,new_node)

    return None
```

Code 4.1: Python graph search algorithm

Code 4.1 is an example of how a search algorithm can be implemented. The first test of this algorithm was with the 8-sliding block puzzle, where there are $\frac{9!}{2} = 181440$ number of states. In fact, there are 9! = 362880 different combinations, but only half of them is possible to solve. This was solved using a heuristic called the Manhattan Distance [34]. The Manhattan distance is the absolute difference of the Cartesian coordinates of the blocks current position and the blocks goal position. Since this is a specialized solver for the 8-puzzle, it is expected to be much faster to solve compared to the generalized PDDL solver. In Figure 4.1 the goal node has been used as a start node and the whole graph has then been explored. Figure 4.1 shows the distribution of the minimum of actions it takes to solve each puzzle. This shows that for some problems it is possible to traverse the whole search space, and when this is done, one can solve each puzzle instantly, because we now have the whole tree. In most of the planning problems, the state space can get very large and it is not possible to find all of the nodes. It is important for large search spaces that a good heuristic is used to guide the search in the right direction.

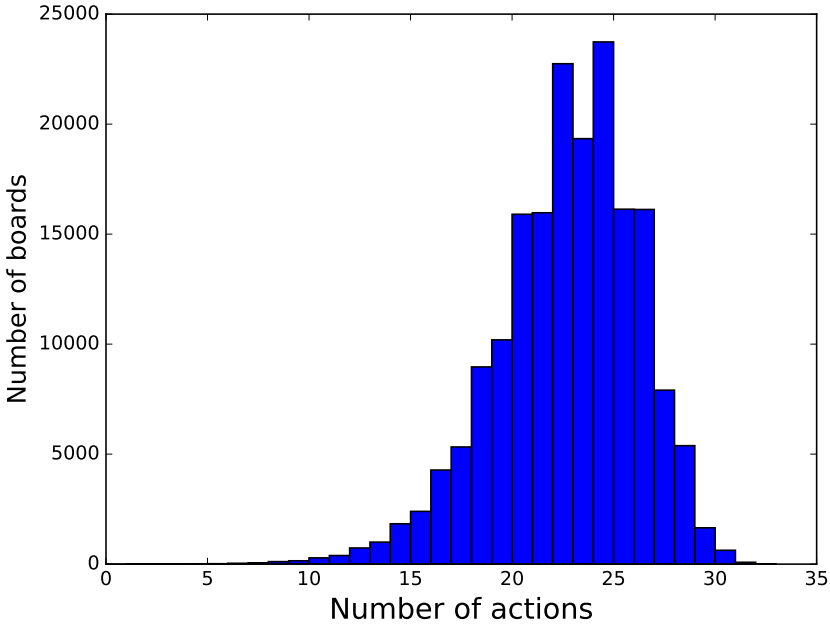


Figure 4.1: Number of actions it takes to solve each puzzle of the 8-puzzle.

For the PDDL solving task, the same solution algorithm in Code 4.1 is used. As mentioned above this is a more generalized solver made to solve problems that can be defined using STRIPS. From Chapter 3 the PDDL files where parsed into two classes. One domain class and one initial state class. The state class must be expanded into calculating the two different heuristics defined in Section 4.1.2.1 and Section 4.1.2.2. The heuristic of the number of remaining subgoals is not so very complicated to implement. The strategy is just to iterate through each subgoal and check if it is in the current state. If not, the heuristic function will be increased by one. Code 4.2 shows how it is implemented in Python.

```
def missing_sub_goal_heuristic(self):
    dist_to_goal = 0
```

```
    for sub_goal in self.goal:
        if not sub_goal in self.state:
            dist_to_goal += 1
    return dist_to_goal
```

Code 4.2: Missing subgoal heuristic

The more complicated heuristic search planner [29] from Section 4.1.2.2 is implemented as seen in Code 4.3. It is a while loop that runs until all the goals are satisfied or the state has stopped expanding. The while loop iterates over each action and checks whether it is applicable or not. If the action is applicable, the relaxed state is expanded with the *add list* from the action. The estimated distance to the goal state is calculated by checking the difference in completed subgoal before and after applying the new states and the difference is multiplied by the depth. This gives the individual cost of completing each subgoal.

```
def ignore_delete_list_heuristic(self):
    state = self.state[:]
    cost = 0

    completed_subgoals = self.get_number_of_completed_subgoals(state)
    depth = 1
    length_goal = len(self.goal)

    old_length = len(state)

    while not completed_subgoals==length_goal:

        add_list = []
        '''Find all applicable actions and their add lists'''
        for action in self.domainclass.actions:
            return_parameters = action.return_possible(state)

            for parameters in return_parameters:

                new_items = action.get_addlist(parameters)
                add_list.extend(new_items)

        completed_subgoals_prev = self.get_number_of_completed_subgoals(
            state)
```

```
        '''Add all of the add lists of the applicable actions'''
        for add_state in add_list:
            if add_state not in state:
                state.append(add_state)

        completed_subgoals = self.get_number_of_completed_subgoals(state)

        cost = cost + depth*(completed_subgoals-completed_subgoals_prev)
        depth = depth + 1

        if old_length==len(state):
            return cost
        old_length=len(state)

    return cost
```

Code 4.3: Relaxed problem ignore delete list and apply all possible actions heuristic Python

4.3 Results

Different solving strategies have now been proposed and discussed. The most basic search algorithms are the BFS and DFS algorithms from Section 4.1.3.1. Further, the resulting sequence of actions from BFS will be the minimum set of actions needed for solving the problem. This gives a good comparison property to be used when looking into other solving strategies as well. The problems that are considered can be seen in the attached files.

In Table 4.1 the different problems that are considered can be seen. Some data are also given to get a feeling about the size of the different problems. All the problems, except the Shakey domain which is from [35], are taken from the *International Conference on Automated Planning and Scheduling (ICAPS)* [36] competition in Artificial Intelligence planning.

Table 4.1: STRIPS problems considered

Name	Number of objects	Size initial state	Available actions
Aircargo	6	10	4
Blocks	10	13	4
Rover 1	25	55	4
Rover 2	13	57	9
Satellite	12	17	5
Shakey	16	40	6

4.3.1 Breadth-First Search

In Table 4.2 the results of the BFS algorithm are seen. The algorithm successfully generates solutions for all of the problems except for the *blocks* domain where the state space expands too much and generates so many states which make the computer use up all of its RAM. The second most complicated problem seems to be the *Rover 1* problem. Actually, the whole state space of the *Rover 1* problem consists of 162875 (see Table 4.3) different states, this means that using BFS one have to search through almost whole of the state space to find a solution. This is not optimal.

Table 4.2: STRIPS problems solved using BFS

Name	Length of solution	Nodes expanded	Nodes in queue	Total runtime[sec]
Aircargo	6	121	17	0.0264
Blocks	Out of memory	528052	715845	144
Rover 1	53	161632	434	140
Rover 2	10	8407	6338	16.7
Satellite	9	429	284	0.185
Shakey	22	8961	154	9.92

In Figure 4.2 the graph structure of a BFS search in the satellite domain is plotted using

Graph-tool [37]. Each node represents a state and the edges represent an action. In the code, the nodes are added when a new state is found and it is connected with its parent state where the action came from. The red thicker edge is the solution path. For a small problem like the satellite problem, the algorithms expand a big set of nodes before it finds a state that satisfies all of the subgoals.

Table 4.3 shows the result of running the BFS algorithm until all reachable nodes from the initial nodes have been visited. The aircargo search space is very small and the time it takes to visit each node does not require a lot of time so the choice of search strategy will not matter that much. But in the blocks domain it exists a lot more states and with the given resources, one can not visit all the states to find a solution.

Table 4.3: Run BFS until all states has been visited or out of memory

Name	State space size	Number of solution states
Aircargo	143	8
Blocks	>1243897(Out of memory)	NA
Rover 1	162875	49
Rover 2	>366590(Out of memory)	NA
Satellite	1856	48
Shakey	9216	1

4.3.2 Depth-First Search

The next algorithm is the Depth-First Search(DFS). The results are given in Table 4.4. The length of the solutions compared to Table 4.2 is significantly longer, but the total runtime has been greatly improved. The blocks domain are again not solvable because the memory runs out. This means that a more sophisticated search algorithm is needed in this case, which will be considered in the next sections. A comparison of the graph structure of DFS and BFS can be found in Figure 4.3. The figures show that the BFS

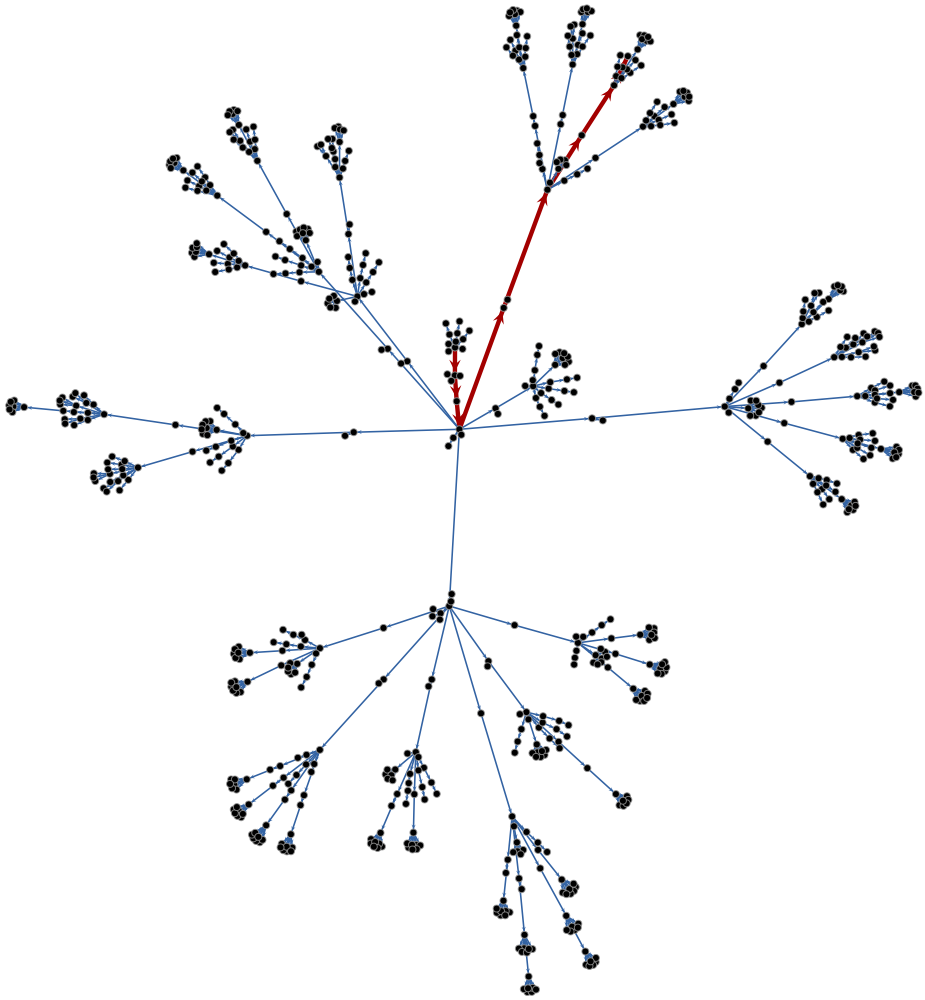


Figure 4.2: BFS graph visualization of Satellite problem

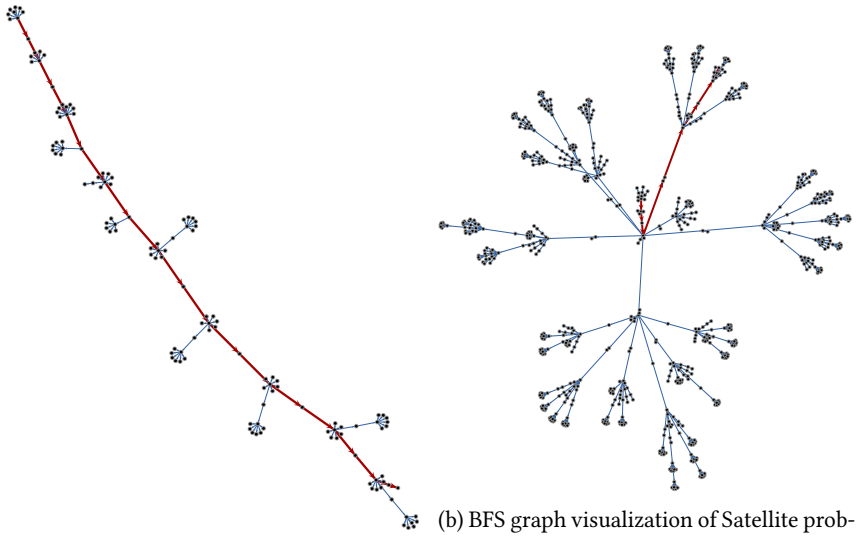
produces a lot more nodes compared to the DFS. The BFS will find the shortest solution but at a greater cost.

Table 4.4: STRIPS problems solved using DFS

Name	Length of solution	Nodes expanded	Nodes in queue	Total runtime[sec]
Aircargo	18	28	53	0.00667
Blocks	Out of memory	19055	35318	27.0
Rover 1	93	141	106	0.123
Rover 2	84	141	415	0.315
Satellite	18	86	43	0.0363
Shakey	92	137	82	0.160

4.3.3 Greedy Best-First Search

Now that some uninformed search strategies have been tested, it is time to test some informed search strategies. In Table 4.5 and Table 4.6 one can see the results of the missing subgoals heuristic from Section 4.1.2.1 and the relaxed problem heuristic from Section 4.1.2.2 respectively. In Table 4.5, a solution for the blocks domain finally appears and at a fast total runtime as well. Compared to Table 4.6 the solution of the blocks problem is very long, but it solves it over six times faster than the relaxed problem heuristic. The greedy best-first search with the missing subgoals heuristic is fast but may find solutions that are unnecessary long. The greedy best-first search with relaxed problem heuristic algorithm finds better solutions at a bit slower runtime this is because the relaxed problem heuristic algorithms must solve the relaxed problem for every new node discovered, which can be time-consuming.



(a) DFS graph visualization of Satellite problem

(b) BFS graph visualization of Satellite problem

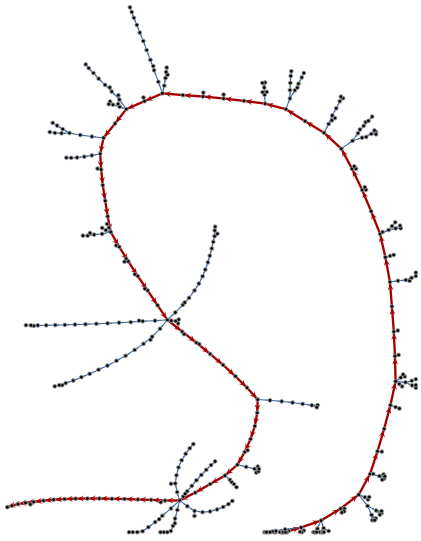
Figure 4.3: Comparison of DFS and BFS search tree

Table 4.5: STRIPS problems solved using greedy best-first search with missing subgoals heuristic

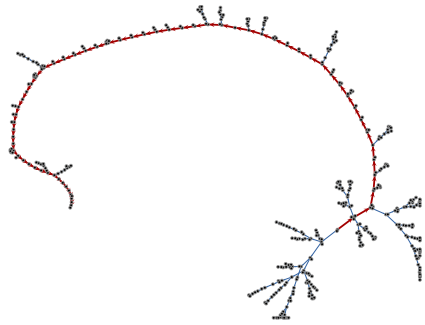
Name	Length of solution	Nodes expanded	Nodes in queue	Total runtime[sec]
Aircargo	6	15	31	0.00392
Blocks	150	2828	2081	0.540
Rover 1	77	269	111	0.235
Rover 2	24	176	206	0.349
Satellite	9	38	37	0.0165
Shakey	40	114	53	0.130

Table 4.6: STRIPS problems solved using greedy best-first search with relaxed problem heuristic

Name	Length of solution	Nodes expanded	Nodes in queue	Total runtime[sec]
Aircargo	6	6	26	0.0185
Blocks	60	250	388	3.41
Rover 1	63	209	217	6.41
Rover 2	10	10	35	0.370
Satellite	9	12	35	0.0600
Shakey	30	86	86	1.35



(a) Greedy best-first search with missing subgoals heuristic graph visualization of Rover1 problem



(b) Greedy best-first search with relaxed problem heuristic graph visualization of Rover1 problem

Figure 4.4: Graph comparison of greedy best-first search heuristics

4.3.4 A*

In Section 4.3.3 the next node to be expanded was only based on the estimate of the remaining distance to the goal. As previously mentioned in Section 4.1.3.3, the A* algorithm also takes the current distance from the start node into account when choosing the next node to expand. In Table 4.7 one can see the results of the A* with the missing subgoals heuristics. Because of the bad informative properties of this heuristic, the algorithm cannot find a solution for the blocks domain.

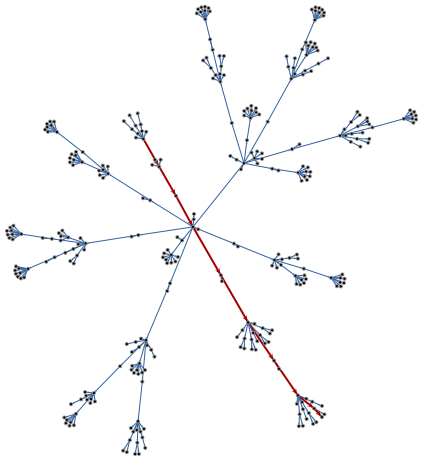
Table 4.7: STRIPS problems solved using A* search with missing subgoals heuristic

Name	Length of solution	Nodes expanded	Nodes in queue	Total runtime[sec]
Aircargo	6	69	48	0.0170
Blocks	Out of memory	536819	718666	153
Rover 1	53	159823	655	142
Rover 2	10	2552	2623	4.96
Satellite	9	132	191	0.0576
Shakey	22	4873	1220	5.41

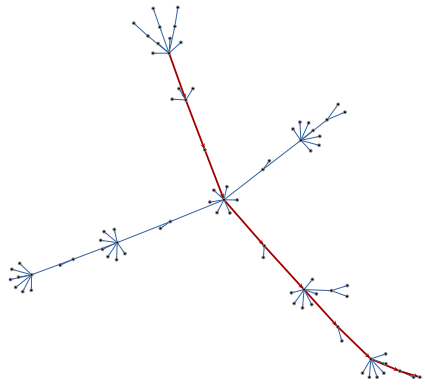
Table 4.8 shows the results of the A* with the relaxed problem heuristic. It finds the solution of all the given example problems, but for the Blocks and Rover1 domains, the total runtime is very high compared to the other search strategies. If the goal is to find a short solution where there are no time constraints, this strategy will be a good one. Since the relaxed problem heuristic is not admissible one can see that that in some cases it will not find the optimal solution. The optimal solution of Shakey is 22, but with the relaxed problem heuristic, the solution has length 26. As one can see in Figure 4.5, the missing subgoals heuristic leads the solver to branch out a lot more than the relaxed problem heuristic does. This can be evidence on that the relaxed problem heuristic is better at estimating the cost.

Table 4.8: STRIPS problems solved using A* search with relaxed problem heuristic

Name	Length of solution	Nodes expanded	Nodes in queue	Total runtime[sec]
Aircargo	6	24	62	0.0488
Blocks	40	54539	94496	906
Rover 1	53	38704	8522	754
Rover 2	10	47	125	1.61
Satellite	9	20	59	0.0962
Shakey	26	839	705	14.0



(a) A* search with missing subgoals heuristic graph visualization of Satellite problem



(b) A* search with relaxed problem heuristic graph visualization of Satellite problem

Figure 4.5: Graph comparison of A* search heuristics

4.3.5 Weighted A*

In Section 4.3.3 the runtime was good, but the length of the solutions were in some cases longer than necessary and in Section 4.3.4 the length of the solutions were good, but the runtime became too long in some of the problems. With the weighted A* (WA*), which is explained in Section 4.1.3.3, one can choose by setting a weight w, if one wants more greedy best-first search behavior or one can choose to get more of the A* behavior where the length of the solution is more important. This means that w is a tuning parameter and is chosen based on the requirements of the task at hand. The results in Table 4.9 and Table 4.10 have the weight $w = 100$. This can be seen as a greedy best-first search where it picks the node with the lowest depth when there are ties in the cost. The results in Table 4.9 are satisfactory. All the problems are solved in under 0.5 seconds and the length of the solutions have a length that is satisfactory as well.

Table 4.9: STRIPS problems solved using weighted A* search with missing subgoals heuristic

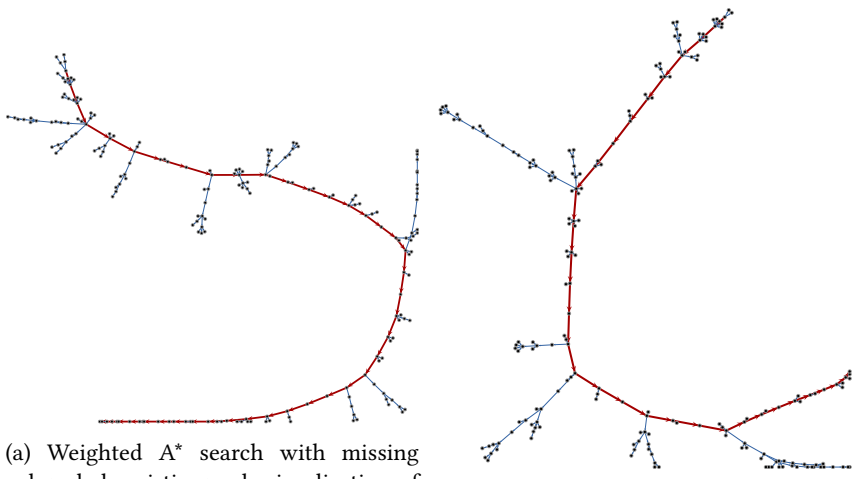
Name	Length of solution	Nodes expanded	Nodes in queue	Total runtime[sec]
Aircargo	6	22	36	0.00560
Blocks	44	141	242	0.0398
Rover 1	65	386	126	0.333
Rover 2	10	79	127	0.156
Satellite	9	34	33	0.155
Shakey	40	145	58	0.167

Table 4.10 shows the results with the relaxed heuristics. Here the solver will again use some time to find a solution to the more difficult problems. The length of the solutions is not so much better either.

Now that many different search strategies have been implemented and tested one can see that the most promising is the weighted A* algorithm. The missing subgoals

Table 4.10: STRIPS problems solved using weighted A* search with relaxed problem heuristic

Name	Length of solution	Nodes expanded	Nodes in queue	Total runtime[sec]
Aircargo	6	6	26	0.0180
Blocks	60	259	387	3.19
Rover 1	61	212	223	6.45
Rover 2	10	10	35	0.369
Satellite	9	12	35	0.0595
Shakey	30	97	97	1.54



(a) Weighted A* search with missing subgoals heuristic graph visualization of Shakey problem

(b) Weighted A* search with relaxed problem heuristic graph visualization of Shakey problem

Figure 4.6: Graph comparison of weighted A* search heuristics

heuristics gave better results than expected. Some of the examples used have multiple independent subgoals where the heuristic makes the solver focus on the actions that make the state satisfy the subgoals. If the goal state was only one subgoal, this heuristic would not perform much better than a breadth-first search. The reason the relaxed problem heuristic is so slow may be of non-optimal programming in the development phase. The search for applicable actions is text-based and the algorithm finding the actions is a recursive for loop with a runtime based on the size of the current state. The difficult problems tend to have larger states which normally generates many applicable actions which in turn generates many new states where one need to run the heuristic function on. I.e. the bottleneck of the code is the function that finds the applicable action and the relaxed problem heuristic uses this function as well when solving the relaxed problem which makes the heuristic solver slow.

Chapter 5

Planning and Replanning for a simulated robotic system

In the previous chapter different solving algorithms were tested on different problems from the ICAPS competition [36]. In this chapter, the solver will be further explored with some examples.

5.1 YouBot

In this example, the KUKA YouBot [38] will be used in a simulation environment to solve two different planning problems. The first one is the Tower of Hanoi and the second is to rearrange stacks of blocks. The YouBot, seen in Figure 5.1, is a five degrees of freedom robotic manipulator with a two finger gripper on top of an omnidirectional base. Five degrees of freedom means here that the robotic manipulator has five joints. All the joints are controllable. Omnidirectional means that the robot can move in all directions regardless of its orientation.

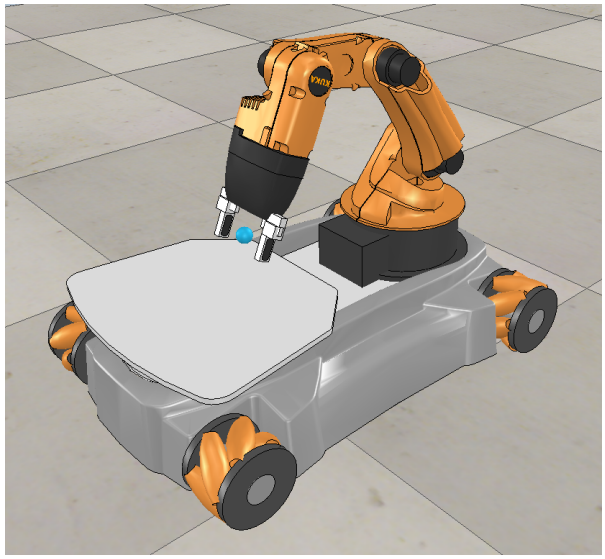


Figure 5.1: Picture of the KUKA YouBot

The simulation tool used to simulate is called V-REP [3]. V-REP is a virtual robot experimentation platform. In this example, the education version of V-REP is used. V-REP also has an integrated development environment which means that one can do development in V-REP via the Lua programming language.

5.1.1 Tower of Hanoi

Tower of Hanoi is a game which consists of a given number of disks of increasing sizes. The disks can be threaded down onto three different sticks. It is possible to move a disk if it is on the top of the stack or is the only disk on the stick. The disk can only be moved onto a disk which is bigger than itself or moved to a stick if the stick has no disk on it. The initial puzzle can be seen in Figure 5.2. The puzzle has three disks, and the goal is to move the stack, one disk at the time, to the opposite stick. In this case, it will be three disks. The minimum number of moves is $h_n = 2^n - 1$ where n is the

number of disks [39]. This means that the minimum number of moves in this example is 7.

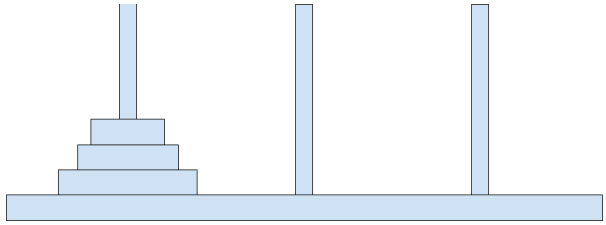


Figure 5.2: Picture of the Tower of Hanoi puzzle

This problem was chosen because it is already a given example in the V-REP installation for the YouBot. In the original example, the solution is hardcoded Lua code written in V-REP . This means that the existing script for the solution will be rewritten into taking orders from the STRIPS solver that was made in Python in the previous chapters. The Python script must have a way to communicate with the embedded Lua script. V-REP has a remote API for Python. The remote API cannot communicate directly with the Lua script, which means that the example and the controllers that have already been written, must be rewritten in Python. This will be time-consuming or maybe not possible since the API has limited functionality compared to the embedded Lua script.

Thankfully, V-REP also supports the Robot Operating System (ROS) [40]. ROS is not really an operating system as its name may state, but a flexible framework for writing robot software. ROS has a collection of libraries, tools and conventions that are meant to simplify the complex process of creating robust and good robot software. Tasks that can be simple for humans, can be complex and hard for a robot. ROS was created so that everyone with knowledge of different aspects robotic can contribute to ROS and at the same time make use of resources others have made. In such ways, ROS is always in development and encourage groups to collaborate to make ROS better.
The features that will be used from ROS in this case, is the publisher and subscriber functionality. This functionality allows sending messages between the Lua script and

Python. The publisher publishes the message to a ROS topic. ROS topics are named buses over which ROS nodes exchange messages [41]. A ROS node is a process that performs computation and is combined with other ROS nodes into a graph and enables them to communicate with each other [42]. The subscriber subscribes to a published topic. This way one can create a ROS node on the V-REP side and one ROS node at the Python side. Then declare a publisher for Python and a subscriber for V-REP which makes it possible to exchange information between these two platforms. An example of how to publish a message is seen below in Code 5.1. The publisher declares that the *action_node* is publishing to the *do_action* topic using the message type String.

```
import rospy
from std_msgs.msg import String

pub = rospy.Publisher('do_action', String, queue_size=10)
rospy.init_node('action_node', anonymous=True)
msg = 'hello world'
pub.publish(msg)
```

Code 5.1: ROS publisher in Python

In Code 5.2 a subscriber in Lua is seen. The subscriber subscribes to the *do_action* topic and every time a message is published to the topic, the subscriber will receive it and the *subscriber_callback* function will be run with the message as the argument. In this case, the message will be an action represented as a string.

```
subscriber=simROS.subscribe('/do_action','std_msgs/String','
    subscriber_callback')
function subscriber_callback(msg)
       action = split(msg.data," ")
end
```

Code 5.2: ROS subscriber in Lua

The approach in sending the desired actions is to send each action step by step from Python as each previous action is completed. Another approach is to send the entire plan right away, but it is desirable to keep the Lua script as low level as possible and do the most of the planning and execution from Python. This means that a subscriber

must be added to the Python side and a publisher must be added to the Lua V-REP side. This is to handle the confirmation messages which states that the action has been completed.

5.1.1.1 PDDL

The Tower of Hanoi domain contains only one action, which is the *(move ?from ?to)* action. The problem file is automatically generated using a Python script that has been made for this purpose. The code only needs to know the number of disks and the path for where to save the generated file. The PDDL files and the code for making the problem file can be found in the attached files.

5.1.1.2 Implementation

Figure 5.3 shows the start position of the Tower of Hanoi. In the original Tower of Hanoi there are disks which are threaded down onto sticks. In this case, the robot will most likely struggle to deal with the practicalities of the original problem. In the example given in V-REP the problem is represented with blocks instead. As one can see, each color represents a disk and one has to move all the blocks with the same color at the same time such that it qualifies as a Tower of Hanoi problem. The actions that the script uses are:

- *pickupBoxFromPlace(boxHandle, pickupConf)*
  This function picks up the box described by the *boxHandle* which are initialized at the start of the script. For example *yellowBox2=sim.getObjectHandle('yellowRectangle2')* will declare a box handle for the second yellow box. The *pickupConf* argument is the desired orientation of the end effector when picking up the box.
- *dropToPlace(placeHandle, shift, verticalPos, startConf, noVerticalArmForUpMovement)*

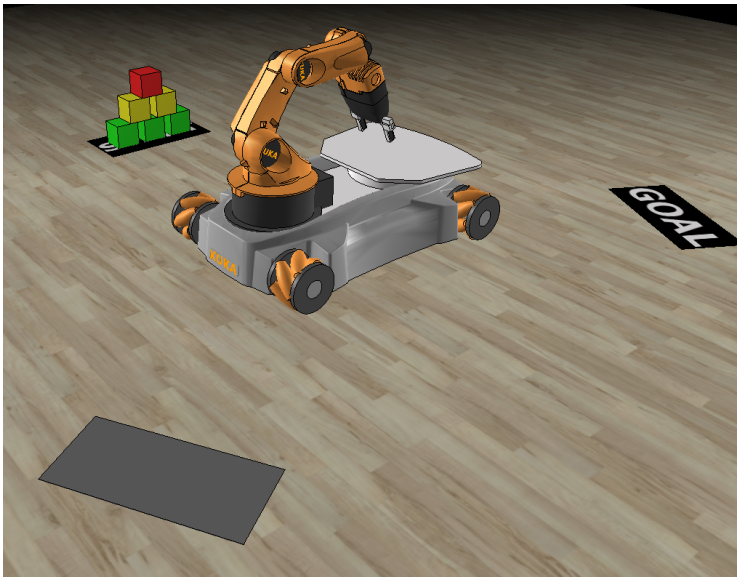


Figure 5.3: YouBot and the Tower of Hanoi

The task of this function is to drop an already picked up box onto a desired spot. The *placeHandle* is a handle for one of the rectangles seen in Figure 5.3 which are used as the three sticks in the original problem. The shift argument declares how much the placement of box shall be shifted left or right relative to the rectangle center. *verticalPos* is the *dropheight* of the box. *startConf* is the initial configuration of the manipulator. The *noVerticalArmForUpMovement* is not used.

- *dropToPlatform(platform)*
  This is used when the robot is supposed to drop a held box onto its platform. The YouBot platform has three preassigned spots for placing boxes. The *platform* arguments states where the box is to be dropped.

- *pickupFromPlatformAndReorient(boxHandle)*
  This function picks up the desired box and drops it on the ground and then picks it up again. This is because the robot arm can not pick the box properly up from its platform. It therefore has to reorient its grip before placing the box.

The V-REP side, which is supposed to be as low level as possible, is only going to translate the desired action from the Python side. The action can for example look like this: *pickupBoxFromPlace redBox1 pickup1*. This makes it possible to effectively handle each order as seen in Code 5.3. The objects, such as the boxes and platform, are mapped in a dictionary called *objectDict*.

```
if message[1]== 'pickupBoxFromPlace' then
    pickupBoxFromPlace(objectDict[message[2]],objectDict[message[3]])
    message = ''
    simROS.publish(publisher,{data='action_completed'})
elseif message[1] == 'dropToPlatform' then
       dropToPlatform(objectDict[message[2]])

...
```

Code 5.3: Action request handled in Lua

The task of the Python script is to solve the planning problem and translate the Tower

of Hanoi *move* action from the solution, into orders that can be used in Lua to control the YouBot . The *move* action has the parameters *disk*, *from* and *to*. The complete *domain.pddl* file is given in the attached files. An important thing to keep in mind here is that the move action for each disk will be different. To move the green disk, which consists of the three green blocks, the YouBot must first place two of them on its platform before it can place them somewhere else. Compared to each other, the process of moving the red box takes two actions, while moving the green boxes takes ten actions. After the solver has solved the planning problem, the plan is iterated through and translated into YouBot actions, as seen in the example for moving the red box in Code 5.4. The code also keeps track of the height of each stack, such that the YouBot drops the box at the right height.

```
if action[1] == 'DISK1':
    self.plan.extend(self.move_red(self.name_map[action[2]],self.name_map[
        action[3]]))
...
def move_red(self,place_from,place_to):
    self.name_map['DISK1'] = place_to
    self.heights[place_to] += 1
    self.heights[place_from] -= 1
    dropheight = 'dropHeight'+str(self.heights[place_to])

    actions = []
    actions.append(['pickupBoxFromPlace redBox1 pickup1'])
    actions.append(['dropToPlace '+place_to+' middle '+dropheight+' pickup1
        '])
    return actions
```

Code 5.4: Python code for generating actions to move the red box

When the list of actions is completed, the ROS publisher will publish each action, step by step as they are completed. The code used for this is seen in Code 5.1. Figure 5.4 show the result after the plan has been executed. The video of the execution can be found in the attachment.

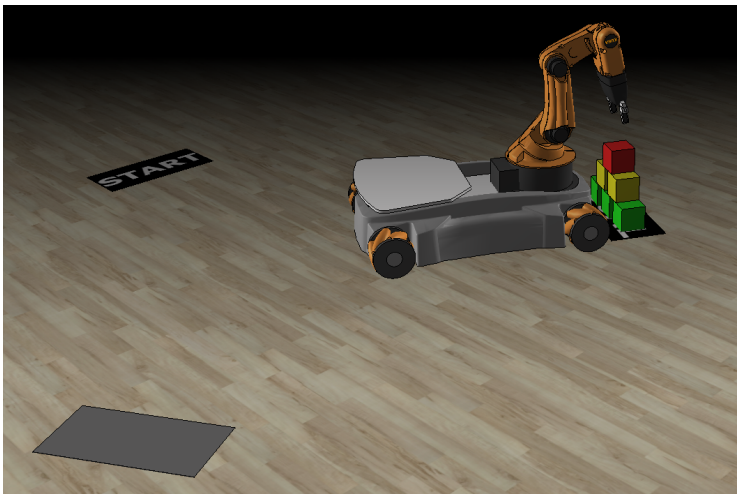


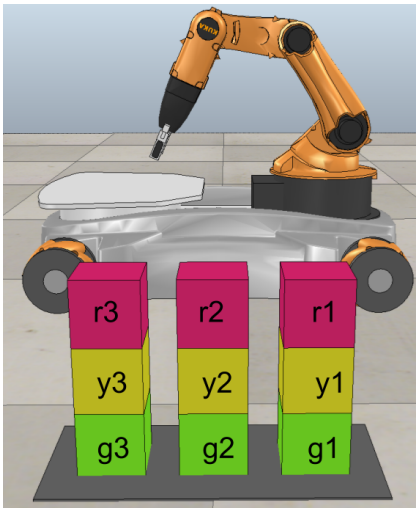
Figure 5.4: YouBot with completed Tower of Hanoi

5.1.2 Restack of blocks

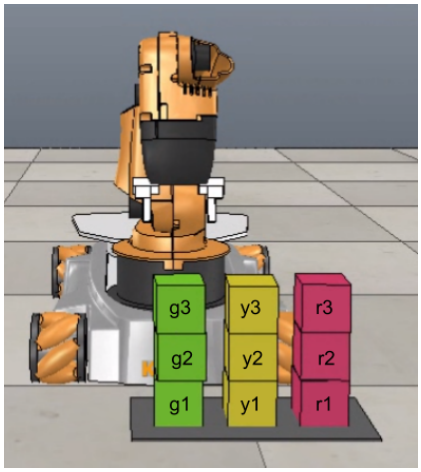
The previous Tower of Hanoi example was used as a transition to this problem, which is a bit more complicated in regard to planning. Also, most of the framework made above can be used here. The problem can be seen in Figure 5.5. The goal is to rearrange three stacks of blocks into a wanted pattern. In Figure 5.5 the goal is to sort the stacks such that is only contains one color. The order of the boxes must also be taken into consideration. The YouBot can store three boxes on its back platform.

The same actions used in Tower of Hanoi is used here with some tweaks to the Lua script. The changes are that the robot does multiple actions at the same time, such as moving to the desired location and rearranging the orientation of the manipulator. In some cases, the robot would knock over a stack because it had just placed a box on the ground and the next action was to pick up a box at another stack. The fix was to make the robot wait until the manipulator was the right configuration such that it would not knock over a stack. Another fix was made to the placement of blocks on its platform. The initial code from the Tower of Hanoi example did not handle having

three blocks at the same time there very well. The fix was made adjusting the position of each joint to try to spread the boxes more over the platform, with varying results. In the attached video file on can see that the robot struggles with placing the blocks on its platform.



(a) The initial placement of three stacks



(b) The goal state for the three stacks

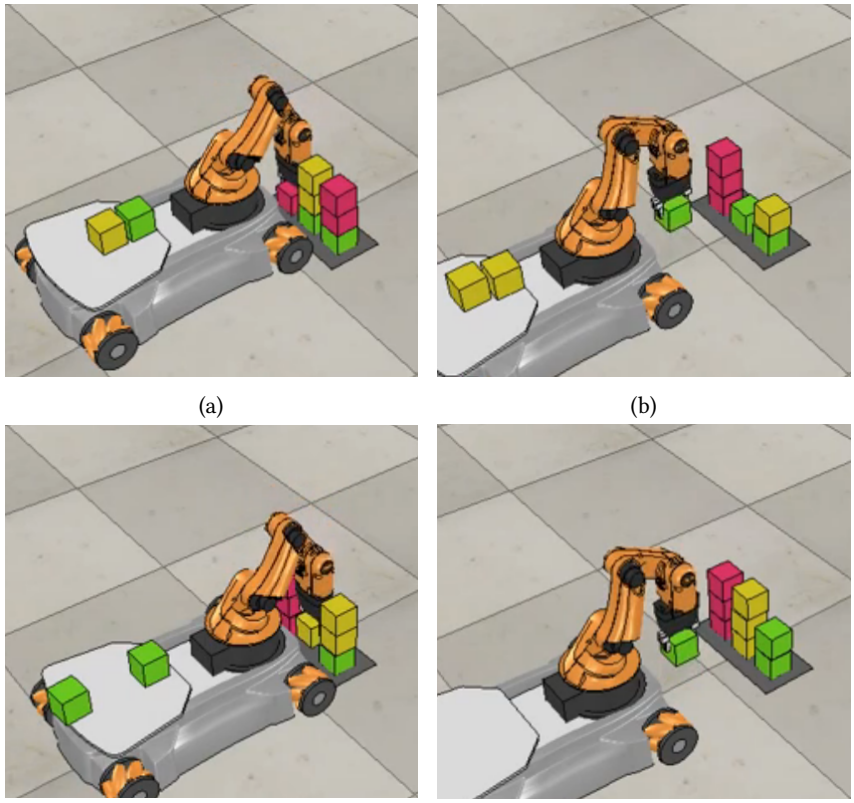
Figure 5.5: The initial stacks and the goal stack

5.1.2.1 PDDL

The technical details were dealt with in the previous section which means the biggest task here is to create the PDDL files. Another thing to keep in mind is that each stack has a max height of three and to make sure that a block is not placed at the uppermost position when there are no blocks underneath to support it. The resulting PDDL-files can be found in the attached files.

5.1.2.2 Results

Figure 5.6 shows four screenshots of the simulation while it is executing. The solution that the solver finds has 56 actions and takes approximately one second to find. The solver that is used here is the Weighted A* with the missing subgoals heuristics. Since the goal state has many subgoals, the solver can focus on the path that takes the state one step closer to the solution. The goals are not independent e.g. if the top box is at the right place, but the boxes underneath are not, the top box must still be moved to complete the problem, which may affect the computation time.



(a) (b)

(c) (d)

Figure 5.6: Snapshots of the restack simulation

5.2 Replanning

In Section 2.2 it was stated that the planning system had to be deterministic with no uncertainty. In the real world, this is not the case. An interesting approach would therefore to explore the opportunities of planning when the known domain changes or the entire domain is not known. One way to replan is simply to replan from scratch [43]. This can be done by updating the *problem.pddl* file to the new changed state and then make a new plan based on the new knowledge about the world. This way, the algorithm will assume that it knows everything about the domain and plan based on that. When new things are discovered, the domain will be updated to a new current view of the domain and plan accordingly.

In Figure 5.7 the flowchart of a replanning algorithm is presented.

5.2.1 Implementation

Consider a domain where a robot can move within a grid. The grid represents a room where there are several obstacles to handle. The goal of the robot is to get to the goal tile, which in this case is a door. The tiles in the grid can contain two kinds of objects. The first kind of object can be picked up by the robot and placed on an empty tile. The other kind of object cannot be moved and the robot must find a way around it. In the initial state, one can decide to make the unmovable objects "invisible", such that in the initial state, it is handled as a moveable object, but when the robot tries to pick it up, it fails. The robot must then replan a route around it.

This problem can be represented using STRIPS. The possible actions in this domain will be *pick-up*, *put-down* and *move*. The *domain.pddl* and the *problem.pddl* file can be seen in the attached files. Unlike the domain file, the problem file is automatically generated. This means that it is possible to generate a grid with desired measurements and also be able to place obstacles around the grid. This makes it easy to test for different problems in the domain. It is also needed to be able to construct the file in

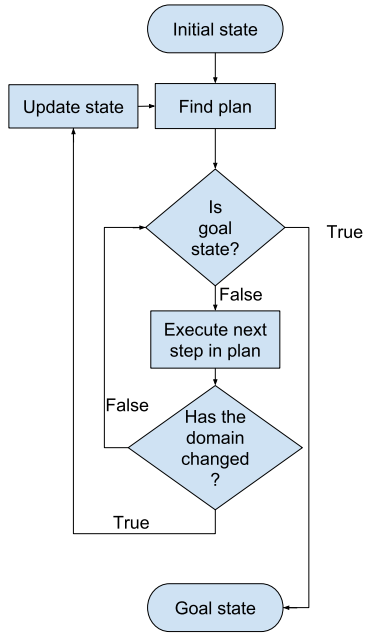


Figure 5.7: Replanning flow chart

order to do replanning. The PDDL problem file is created with the *Domain_rob_to_door* class in the *domain_rob_to_door.py* file in the attachments.

To do the *(move ?from ?to)* action, the predicate *(can-move ?from ?to)*, must be true. One way to automatically generate all the possible movement options is to generate an adjacency matrix($n \times n$) where $n = length \cdot width$ and where the width and length are the size of the grid domain for the robot. The possible move actions for the robot is up, down, right and left. This means for a grid with size 2×2 the adjacency matrix will look like the matrix seen below in (5.1). Column one means that the robot can move from tile 1 to tile 2 or 3. The second column means that it is possible to move from tile 2 to tile 1 or 4, and so on. With this in place, it is easy to add the *can-move from to* atoms to the initial state.

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \tag{5.1}$$

The *problem.pddl* file is created with a list of strings where each element of the list represents one line in the file. For example the obstacle objects are just iterated through and added to the list as seen in Code 5.5.

```
obstacle_num = 1
for obstacle in obstacles:
    lines.append('obstacle'+str(obstacle_num))
    obstacle_num += 1
```

Code 5.5: Adding obstacle objects to the pddl file

Just showing the solution can sometimes be a bit abstract and if one is also to do replanning one need to make a module that sends data back if the world has changed. E.g. in this case it will be if the obstacle object cannot be picked up. In Figure 5.8 one can see the simulation of the robot-to-door domain. In this case, the grid consists of 5×4 tiles or waypoints. The *r* stands for the robot, the *O*s are movable(or unmovable)

objects. The *D* is the door i.e. the desired position. The robot can either pick up the obstacles or go around them. The empty tiles are where the robot can move. Unmovable objects are illustrated as vertical lines |, but cannot be seen in the initial state because they have not been discovered yet. The goal is to get to the goal node with an empty hand.

```
Initial state
[' r ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', ' D ']
```

Figure 5.8: Initial state of the robot to door world

After the problem file has been made it will be solved by the solver. In the case above, the initial solution found consists of 13 actions. Each action is then sent to the *do_action* function. This function attempts to do the action. If the action is successful it will update the simulation and return *true*. If it is not successful it will update the *problem.pddl* file and return *false*. When a *false* is returned, the solver will find a new plan by solving the updated problem. This continues until the robot reaches the goal tile, or it can not reach the target. In Figure 5.9 one can see the start and end of the solution. The total number of actions was 37, including the unsuccessful actions, which is considerably more than the initial 13. The waypoint numbering reads left to right from the top down.

5.3 YouBot replanning

In Section 5.2 the foundations for replanning was made. In Section 5.1, the framework for planning and execution for the YouBot was implemented. In this section, the task is to combine the work made in these two previous sections to do replanning with the YouBot . The task at hand will be most similar to the replanning example in Section 5.2,

```
pickup ROBOT OBSTACLE4 WAYPOINT0 WAYPOINT1
[' r ', ' | ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', ' D ']
Can not pick up. Replanning...
Replanning done


pickup ROBOT OBSTACLE2 WAYPOINT0 WAYPOINT5
[' rO', ' | ', ' O ', ' O ', '   ']
['   ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', ' D ']


move ROBOT WAYPOINT0 WAYPOINT5
['   ', ' | ', ' O ', ' O ', '   ']
[' rO', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', ' D ']


put-down ROBOT OBSTACLE2 WAYPOINT5 WAYPOINT0
[' O ', ' | ', ' O ', ' O ', '   ']
[' r ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', ' D ']


pickup ROBOT OBSTACLE5 WAYPOINT5 WAYPOINT6
[' O ', ' | ', ' O ', ' O ', '   ']
[' r ', ' | ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', ' D ']
Can not pick up. Replanning...
Replanning done


pickup ROBOT OBSTACLE1 WAYPOINT5 WAYPOINT10
[' O ', ' | ', ' O ', ' O ', '   ']
[' rO', ' | ', ' O ', ' O ', '   ']
['   ', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', ' D ']


move ROBOT WAYPOINT5 WAYPOINT10
[' O ', ' | ', ' O ', ' O ', '   ']
['   ', ' | ', ' O ', ' O ', '   ']
[' rO', ' O ', ' O ', ' O ', '   ']
[' O ', ' O ', ' O ', ' O ', ' D ']
```

(a) First seven steps of the robot to door replanning problem

```
put-down ROBOT OBSTACLE8 WAYPOINT2 WAYPOINT7
[' O ', ' | ', ' r ', ' O ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' O ', ' O ', ' | ', ' D ']


pickup ROBOT OBSTACLE12 WAYPOINT2 WAYPOINT3
[' O ', ' | ', ' rO', '   ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' O ', ' O ', ' | ', ' D ']


move ROBOT WAYPOINT2 WAYPOINT3
[' O ', ' | ', '   ', ' rO', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' O ', ' O ', ' | ', ' D ']


put-down ROBOT OBSTACLE12 WAYPOINT3 WAYPOINT2
[' O ', ' | ', ' O ', ' r ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' O ', ' O ', ' | ', ' D ']


move ROBOT WAYPOINT3 WAYPOINT4
[' O ', ' | ', ' O ', '   ', ' r ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' O ', ' O ', ' | ', ' D ']


move ROBOT WAYPOINT4 WAYPOINT9
[' O ', ' | ', ' O ', '   ', '   ']
[' O ', ' | ', ' O ', ' | ', ' r ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' O ', ' O ', ' | ', ' D ']


move ROBOT WAYPOINT9 WAYPOINT14
[' O ', ' | ', ' O ', '   ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' | ', ' O ', ' | ', ' r ']
[' O ', ' O ', ' O ', ' | ', ' D ']


move ROBOT WAYPOINT14 WAYPOINT19
[' O ', ' | ', ' O ', '   ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' | ', ' O ', ' | ', '   ']
[' O ', ' O ', ' O ', ' | ', ' r ']
```

(b) Last eight steps of the robot to door replanning problem

Figure 5.9: First and last steps of of the robot to door solution

but instead of the uncertainty that the objects are movable or not, there can now appear new boxes in the grid world. Figure 5.10 shows the start of the YouBot and the grid world. There are four boxes already spawned in the grid. The goal for the YouBot is to pick up the red box in the upper right corner and drive it to the upper left corner. The grid is made up by a 5×5 grid which is highlighted in the figure. The YouBot cannot move to tiles that are occupied by a box. Unless it shall pick the box up. In the middle of the simulation, three blue boxes will spawn blocking the path to the goal tile. This will trigger the planner to replan and find a way to get to the goal tile (upper left corner).

Figure 5.10: The initial layout of the grid world

5.3.1 Implementation

5.3.1.1 V-REP - Lua

To make this work in V-REP some additional functionality is needed. The robot can do three things in the given domain: *pick-up*, *put-down* and *move*. From before, the robot has no *move(x,y)* function where (x,y) are coordinates in V-REP . Until now this functionality has been included in the functions presented in Section 5.1.1.2, where the robot moved to the boxes based on the position of the boxes. In this new domain, the robot needs to move from tile to tile given the coordinates. Also, the movement seen in the previous examples are combined with reorientation and translation in one movement. When moving in this grid world, this kind of movement will in most cases cause the YouBot to move into the other tiles where it can crash into some of the boxes or the wall. The *move(x,y)* action will therefore consist of two processes. First, it will reorient where the new orientation is calculated as $\theta = \text{atan2}(dx, dy)$, where dx and dy is the difference between the current position and the new position. atan2 is used to get the angle in the right quadrant. Code 5.6 shows how it is done in Lua.

```
local newOrientation = math.atan2(dx,dy)

sim.setObjectOrientation(vehicleTarget,-1,{0,0,newOrientation})
waitToReachVehicleTargetPositionAndOrientation()

sim.setObjectPosition(vehicleTarget,-1,{x,y,0})
waitToReachVehicleTargetPositionAndOrientation()
```

Code 5.6: Code from the move function in Lua

For the action to pick up a box, the same reorientation/move problem arises when the robot reorients itself at the same time it is moving. It is solved by adding a wait function between the reorientation function and move function. To optimize the time consumption, it is possible to make reorient a STRIPS action as well, but that will make the planning problem more complex and bigger and is not considered in this case. The last function, *put-down(x,y)*, makes the robot to place the box it is holding at coordinates *(x,y)* and then return back to the tile it was previously at. This is

done by three actions in Lua: *move to box→put-down box →move back to previous position.*

5.3.1.2 STRIPS

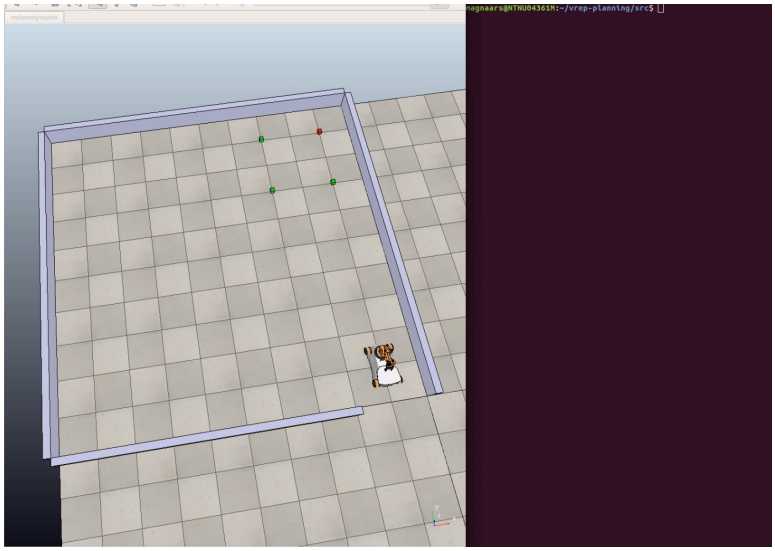
The robot has in this domain three actions. The actions and their preconditions and effects are the same as in Section 5.2 with the exception that when a box is picked up, the robot will be at the position from where the box was picked up. This is because the robot needs to drive to the box when it shall pick it up and it will in most cases be inefficient to drive back. This is handled in Python by adding a *pick-up* action and *move* action to the plan to be sent to V-REP over ROS. The problem file is automatically generated using the initial knowledge of the world. The files can be found in the attachments.

5.3.1.3 Python

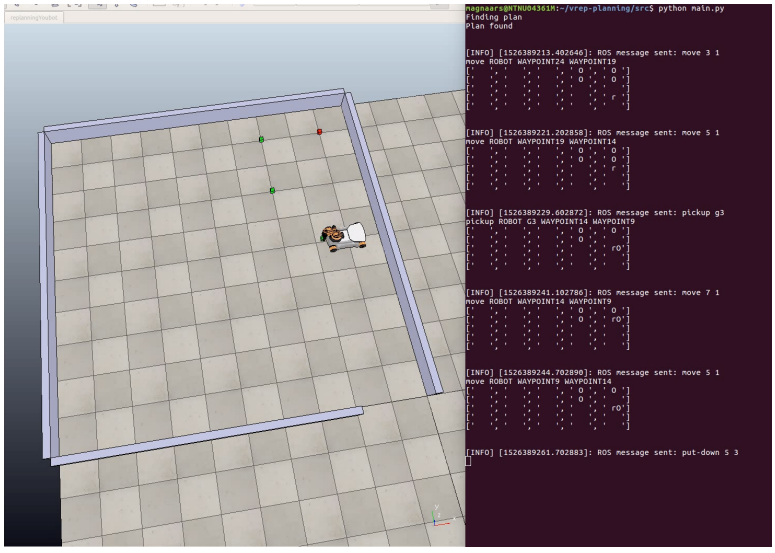
It is in Python most of the changes will be done. Because the domain and problem are almost the same, some of the code used in Section 5.2 will be reused here. The main problem is the real-time challenges that appear, such as the robot must execute the whole plan, and when it needs to replan, the robot can not continue to execute the old plan. When new boxes appear, the Lua script will publish an interrupt message on the already existing ROS topic. When the Python script gets the message, it will stop sending new orders to the YouBot and start to replan. When a new plan is found, the Python script will start sending the new plan back to the YouBot . The replanning happens in the same way as in Section 5.2, by updating the *problem.pddl* file to the current state of the domain and then solve the new problem.

5.3.2 Results

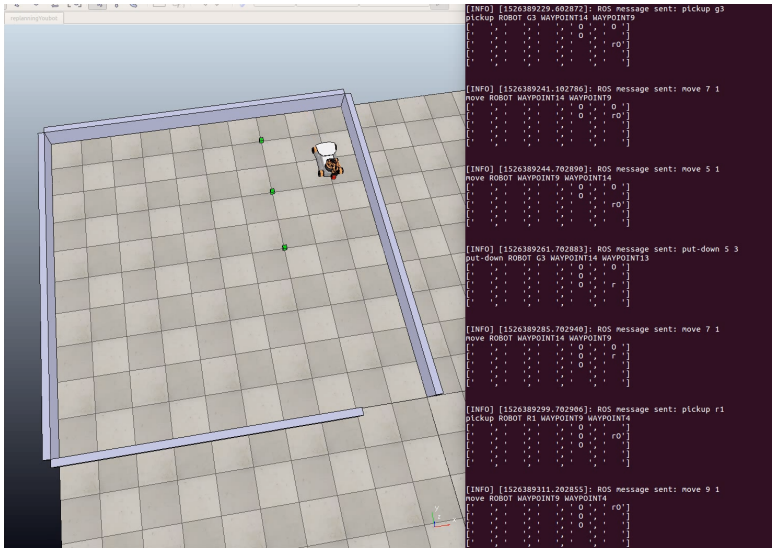
Figure 5.11 shows eight screenshots from the simulation. In the figures both the V-REP simulation is seen and the terminal where information from the Python side is printed as well. First, it displays the string that is published to the ROS topic. Then it shows the STRIPS action that has been executed. The last thing is the visualization of how the current state of the grid world looks like. The symbols are the same as in Section 5.2. In the figures on can see when the YouBot comes close to its goal, three new boxes are spawned blocking its way. A new plan then found and executed and the YouBot gets to the goal position with the red box in its gripper.


magnaars@NTNU04361M:~/vrep-planning/src$

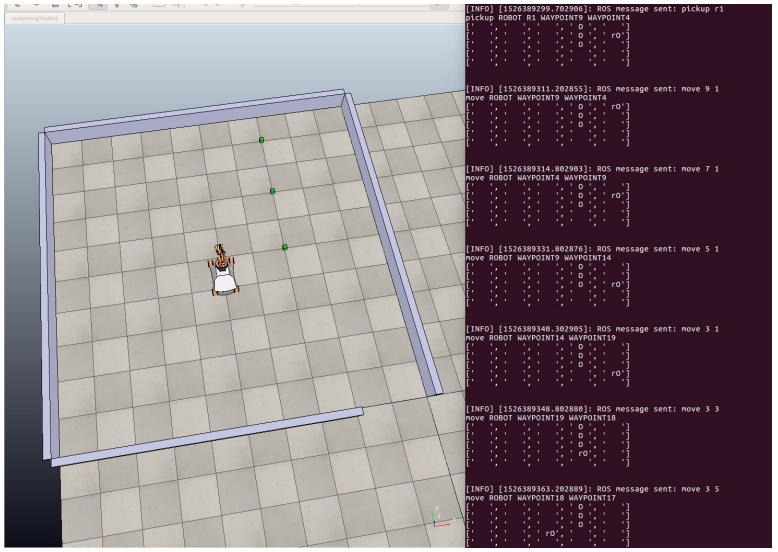
(a)


magnaars@NTNU04361M:~/vrep-planning/src$ python main.py
Finding plan
Plan found
[INFO] [1526389213.402646]: ROS message sent: move 3 1
move ROBOT WAYPOINT24 WAYPOINT19
[INFO] [1526389221.202858]: ROS message sent: move 5 1
move ROBOT WAYPOINT19 WAYPOINT14
[INFO] [1526389229.602872]: ROS message sent: pickup g3
pickup ROBOT G3 WAYPOINT14 WAYPOINT9
[INFO] [1526389241.102786]: ROS message sent: move 7 1
move ROBOT WAYPOINT14 WAYPOINT9
[INFO] [1526389244.702890]: ROS message sent: move 5 1
move ROBOT WAYPOINT9 WAYPOINT14
[INFO] [1526389261.702883]: ROS message sent: put-down 5 3

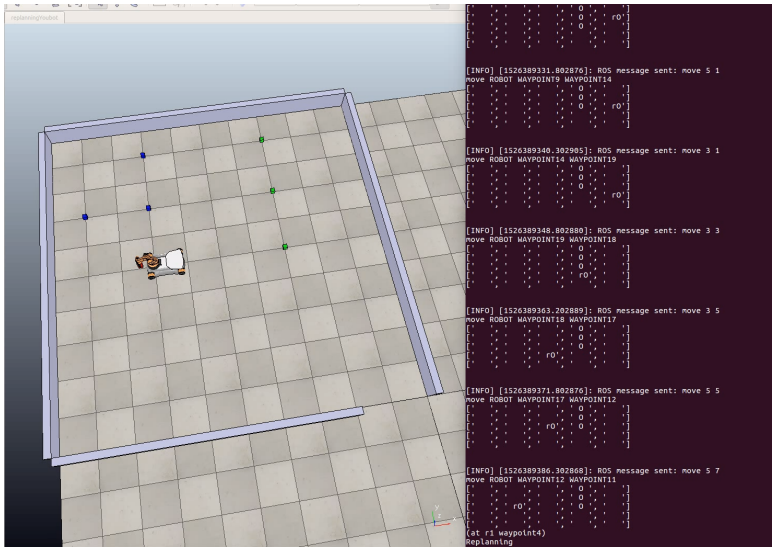
(b)


[INFO] [1526389229.602872]: ROS message sent: pickup g3
pickup ROBOT G3 WAYPOINT14 WAYPOINT9
[INFO] [1526389241.102786]: ROS message sent: move 7 1
move ROBOT WAYPOINT14 WAYPOINT9
[INFO] [1526389244.702890]: ROS message sent: move 5 1
move ROBOT WAYPOINT9 WAYPOINT14
[INFO] [1526389261.702883]: ROS message sent: put-down 5 3
put-down ROBOT G3 WAYPOINT14 WAYPOINT13
[INFO] [1526389285.702940]: ROS message sent: move 7 1
move ROBOT WAYPOINT14 WAYPOINT9
[INFO] [1526389299.702906]: ROS message sent: pickup r1
pickup ROBOT R1 WAYPOINT9 WAYPOINT4
[INFO] [1526389311.202855]: ROS message sent: move 9 1
move ROBOT WAYPOINT9 WAYPOINT4

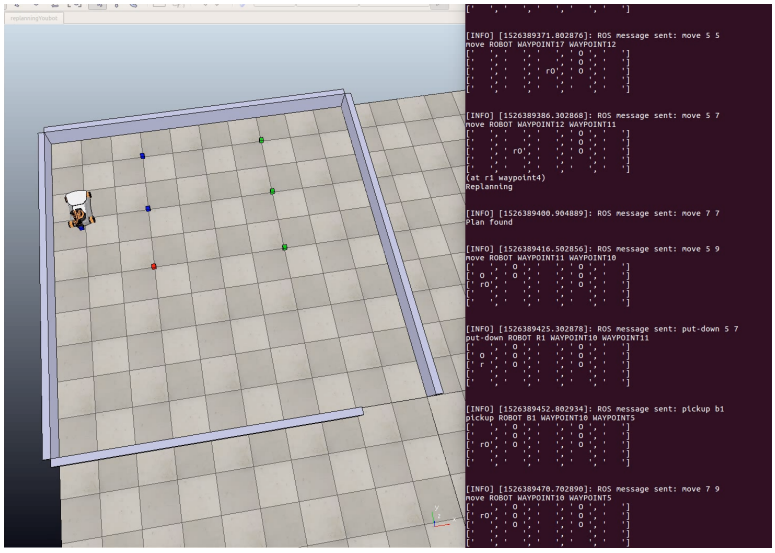
(c)


[INFO] [1526389299.702906]: ROS message sent: pickup r1
pickup ROBOT R1 WAYPOINT9 WAYPOINT4
[INFO] [1526389311.202855]: ROS message sent: move 9 1
move ROBOT WAYPOINT9 WAYPOINT4
[INFO] [1526389314.802903]: ROS message sent: move 7 1
move ROBOT WAYPOINT4 WAYPOINT9
[INFO] [1526389331.802876]: ROS message sent: move 5 1
move ROBOT WAYPOINT9 WAYPOINT14
[INFO] [1526389340.302905]: ROS message sent: move 3 1
move ROBOT WAYPOINT14 WAYPOINT19
[INFO] [1526389348.802880]: ROS message sent: move 3 3
move ROBOT WAYPOINT19 WAYPOINT18
[INFO] [1526389363.202889]: ROS message sent: move 3 5
move ROBOT WAYPOINT18 WAYPOINT17

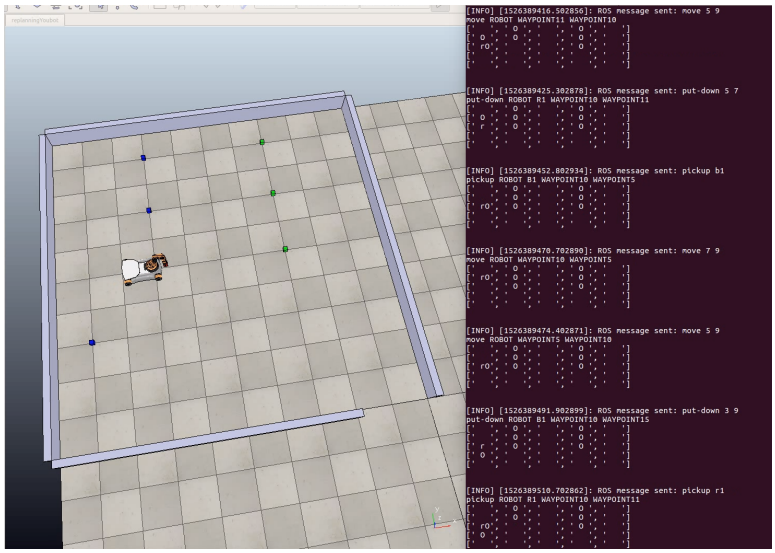
(d)


[INFO] [1526389331.802876]: ROS message sent: move S 1
move ROBOT WAYPOINT9 WAYPOINT14
[INFO] [1526389340.302985]: ROS message sent: move 3 1
move ROBOT WAYPOINT14 WAYPOINT19
[INFO] [1526389348.802880]: ROS message sent: move 3 3
move ROBOT WAYPOINT19 WAYPOINT18
[INFO] [1526389363.202889]: ROS message sent: move 3 5
move ROBOT WAYPOINT18 WAYPOINT17
[INFO] [1526389371.802876]: ROS message sent: move S 5
move ROBOT WAYPOINT17 WAYPOINT12
[INFO] [1526389386.302868]: ROS message sent: move S 7
move ROBOT WAYPOINT12 WAYPOINT11
(at r1 waypoint4)
Replanning

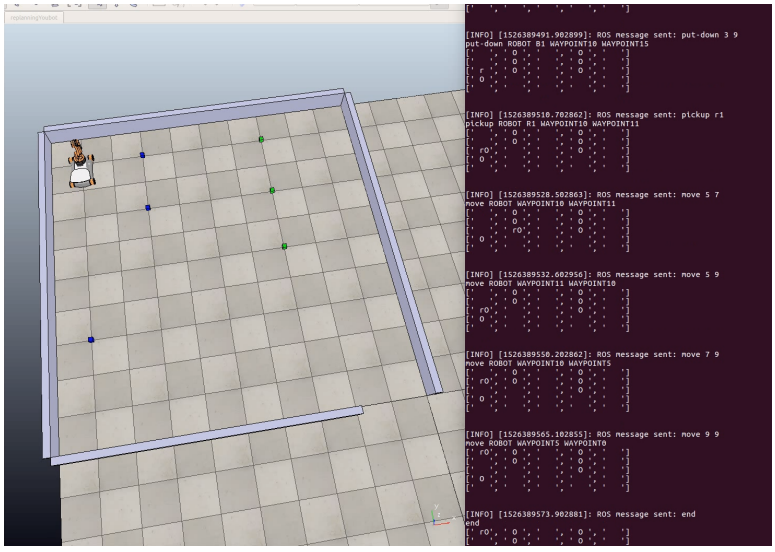
(e)


[INFO] [1526389371.802876]: ROS message sent: move S 5
move ROBOT WAYPOINT17 WAYPOINT12
[INFO] [1526389386.302868]: ROS message sent: move S 7
move ROBOT WAYPOINT12 WAYPOINT11
(at r1 waypoint4)
Replanning
[INFO] [1526389400.904889]: ROS message sent: move 7 7
Plan found
[INFO] [1526389416.502856]: ROS message sent: move S 9
move ROBOT WAYPOINT11 WAYPOINT10
[INFO] [1526389425.302078]: ROS message sent: put-down S 7
put-down ROBOT R1 WAYPOINT10 WAYPOINT11
[INFO] [1526389452.802934]: ROS message sent: pickup b1
pickup ROBOT B1 WAYPOINT10 WAYPOINT5
[INFO] [1526389470.702890]: ROS message sent: move 7 9
move ROBOT WAYPOINT10 WAYPOINT5

(f)



(g)



(h)

Figure 5.11: Snapshots of the YouBot replanning simulation

Chapter 6

Discussion

This chapter includes discussion of the method of the work, the results and potential future extensions.

6.1 Method

One of the goals was to create a heuristic search planner to solve STRIPS problems. This planner was tested with different planning problems mainly from the ICAPS competition [36]. Because of no former planning experience, the author decided to make a planner from scratch. In hindsight, this method may not be the best idea. The parser in Chapter 3 were much more time consuming than originally thought and maybe it might have been more educative to use an already existing parser. This would allow to spend more time on developing the solver and make a more complete solution for the YouBot simulation and planning.

Using STRIPS as the planning language was in this case sufficient enough for the application it was tested on. It facilitated creating the parser such that only conjunctions

are supported, along with that an action only consists of removing or adding atoms from the state. A downside example may be the case of the restacking of blocks with the YouBot where the stacks had a max height of three. In this case, a lot of extra atoms were needed to represent the height and making sure that the boxes were not placed wrong. With ADL, which has more expressiveness, would make the representation better and thus make it easier for the solver.

The KUKA YouBot was the main case study for this paper. V-REP [3] was selected simulation platform, which turned out to be a good choice because of its ease of use. The embedded Lua scripting setup was easy to work with and the overall experience was good.

6.2 Results

6.2.1 Parser

As mentioned above, the parser included a lot more work than expected. One of the main tasks for a planner is to find a solution quickly. All of the code was written in Python, which is not as fast as other programming languages as, for example, C. The task for the parser was also to find the applicable actions for each new state. This is was the part that consumed most of the computational resources. The method used for finding applicable actions was string-based, which is also not an efficient way of doing operations. Because of this, the function had to be rewritten several times and the end result could be better. This problem is similar to, a Constraint Satisfaction Problem(CSP), which is a field of research itself, and outside the scope of this thesis.

6.2.2 Solver

Several algorithms and heuristics were tested and used. The results varied based on the problem and the solver was able to solve the easier problems from the ICAPS competitions fairly fast.

The heuristic search planner, where the ignoring delete list heuristic was used, was successful in finding a solution in few steps. Due to solving a relaxed problem for each new found state, and because of the limiting effect from the slow process of finding applicable actions, made the heuristic method slow, especially were the problem states were large. In such cases, the missing subgoal heuristic would find a solution quicker. Compared to solvers such as Fast Forward, the solver is some levels beneath in performance [44, 45].

6.2.3 YouBot replanning

The results of the replanning for the YouBot were satisfactory. One main goal was to add some kind of uncertainty to the domain. In this case, since the planning was developed from scratch, the main challenge was to set up the framework for enabling to update the problem file to the current state of the domain. In fact, one of the harder problems for the solver was the replanning for when the new blue boxes appeared. This is because the state was relatively large with over 140 atoms, where most of them are adjacency atoms. Because one of the goals that is already satisfied with holding the red box in the gripper and the YouBot must drop the red box in order to clear the path, the solver seems to confuse the heuristic methods used and the solver needs some additional time to solve the problem. In planning theory, this is known as the Sussman anomaly [46] where the robot has to undo a subgoal to complete another.

6.3 Future work

6.3.1 Parser and solver

The program runs only on one thread, so in the future work, one could consider making the parser to use several cores to speed up the process.

The heuristics implemented in this thesis are among the simplest. For a state-space search, there are a lot of other heuristic functions that are available and can be implemented. Also, the solving method can be improved with a better search algorithm [17, 28].

The planner and the YouBot functionality are uncoupled, which means that substituting the planner made in this paper with a state-of-the-art planner can be done if one wants to solve more complex YouBot planning problems efficiently.

The WA* algorithm can yield varying results based on the weights chosen. When there are constraints on the available time for planning, one can use the concept of Anytime Planning. Where one will start with a high weight, w, to get a solution quickly and then use the remaining available planning time to improve the solution by decreasing the weight such that WA* approaches A* characteristics [47, 48, 49, 50, 51].

6.3.2 YouBot replanning

Because of the way action control is handled, the movement of the YouBot in the grid world replanning domain is somewhat piecewise. As one can see in the attached video, the YouBot stops at every waypoint in order to receive the next order. If a plan involves a number of consecutive actions representing respective motions, then a more refined curves path can be generated by making a spline or path to the end waypoint such that the YouBot gets a more smoother movement and a quicker execution. In the implementation of this solution, all actions are considered atomic, which means that an action can not stop once it has started. If one going to include several actions

into one action, one must consider some interruption functionality as well, in case of something happens that has not been planned for.

Further work towards real-life implementation could involve adding a camera, or any kind of perception sensor, which gives information about the grid world. This can be used to add object detection and can then be used by the planner to plan/replan.

Because of the time constraints, there was no time to implement this on a real system. The methodology developed in this work is transferable to other systems and it will not require much to get the planner/replanner to work on other similar robotic systems. As the system is now, it is based on a discretized world split in cells, where the robot has been given no time constraint to find a new plan on how to act on the changes in the domain. If the robot is to act in a more dynamic world where for example humans also are interacting within the robot's domain, replanning may take too long and a kind of reacting behavior must be available to the system. This means that the current plan must be ignored to make place for a react action. To this end, the work produced in this thesis could be well connected to the existing and vast literature on collision avoidance [52, 53].

The method of replanning from scratch may be the easiest and simplest way of doing replanning. Another interesting approach would be to generate two or more plans and if one plan does not work, the algorithm would try to switch plan, which may still be applicable.

6.4 Conclusion

In this thesis, a STRIPS planner has been made and tested with different solution approaches. The focus was to make a planner that could solve simple planning problems at a reasonable time. This has been a difficult task and although one could wish that the resulting implementation was faster, the end result was satisfying. The planner has some work left to do if it is going to be as fast as state-of-the-art planners.

After the planner had reached a satisfactory performance, it was used with a real-time simulation. The simulation was done in V-REP on the KUKA YouBot . The YouBot was easy to work with because V-REP provided basic control functionality which could be adapted into fitting the needs of the task at hand without too many complications. ROS was used to connect the planner written in Python with the embedded V-REP script written in the Lua programming language. The framework built for replanning and execution of the planning domain has a lot of potential and because of the way it has been made, it should be easy to use this in other examples and simulation platforms along with real-world tasks.

Appendix A

Links

The code and videos explained in this thesis are found in the attachment. If the attachment is not available, the content can also be found at this link:
Attachments: `https://drive.google.com/open?id=1KyjKUX_8_hs6XvAS77wivFEIH9gMGBIb`

The videos have also been uploaded to YouTube at:
Hanoi - `https://www.youtube.com/watch?v=wkwClfFRocU`
Restack - `https://www.youtube.com/watch?v=JtnVpeEi1zw`
Replan - `https://www.youtube.com/watch?v=TJiHtSCd2Rg`

The source code can also be found at these GitHub repositories:
Planner - `https://github.com/Aarskog/Planning`
V-REP - `https://github.com/Aarskog/vrep-planning`

References

[1] Kanna Rajan and Alessandro Saffiotti. Towards a science of integrated AI and Robotics, 2017.

[2] Python 2.7.0 release | python.org. `https://www.python.org/download/releases/2.7/`. (Accessed on 05/27/2018).

[3] Coppelia Robotics. V-rep. `http://www.coppeliarobotics.com/`, 2018. [Accessed: 7-May-2018].

[4] Ros.org | powering the world's robots. `http://www.ros.org/`. (Accessed on 05/27/2018).

[5] Nils J Nilsson. Shakey the robot. Technical report, SRI INTERNATIONAL MENLO PARK CA, 1984.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[7] Paul VC Hough. Machine analysis of bubble chamber pictures. In *Conf. Proc.*, volume 590914, pages 554–558, 1959.

[8] Peter E Hart. How the Hough transform was invented [DSP History]. *IEEE Signal Processing Magazine*, 26(6), 2009.

[9] World robotics report 2016 - international federation of robotics. https://ifr.org/ifr-press-releases/news/world-robotics-report-2016. (Accessed on 05/25/2018).

[10] Artificial Intelligence Swarms Silicon Valley on Wings and Wheels - The New York Times. https://www.nytimes.com/2016/07/18/technology/on-wheels-and-wings-artificial-intelligence-swarms-silicon-valley.html. (Accessed on 05/28/2018).

[11] Alessandro Saffiotti, Kurt Konolige, and Enrique H Ruspini. A multivalued logic approach to integrating planning and control. *Artificial intelligence*, 76(1-2):481–526, 1995.

[12] Sebastian Thrun, Maren Bennewitz, Wolfram Burgard, Armin B Cremers, Frank Dellaert, Dieter Fox, Dirk Hahnel, Charles Rosenberg, Nicholas Roy, Jamieson Schulte, et al. Minerva: A second-generation museum tour-guide robot. In *Robotics and automation, 1999. Proceedings. 1999 IEEE international conference on*, volume 3. IEEE, 1999.

[13] Nicola Muscettola, P Pandurang Nayak, Barney Pell, and Brian C Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[14] Kanna Rajan, Douglas Bernard, Gregory Dorais, Edward Gamble, Bob Kanefsky, James Kurien, William Millar, Nicola Muscettola, Pandurang Nayak, Nicolas Rouquette, et al. Remote agent: An autonomous control system for the new millennium. In *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 726–730. IOS Press, 2000.

[15] Pl-plan: A java open-source ai planner. http://www.philippe-fournier-viger.com/plplan/index.php. (Accessed on 05/29/2018).

[16] Optaplanner - constraint satisfaction solver (java™, open source). https://www.optaplanner.org/. (Accessed on 05/29/2018).

[17] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

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

[19] John McCarthy. Situations, actions, and causal laws. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1963.

[20] John McCarthy and Patrick J Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Readings in artificial intelligence*, pages 431–450. Elsevier, 1981.

[21] Raymond Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, 27:359–380, 1991.

[22] Fangzhen Lin. Situation calculus. *Foundations of Artificial Intelligence*, 3:649–669, 2008.

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

[24] Leora Morgenstern. The problem with solutions to the frame problem. *The Robot's Dilemma Revisited: The Frame Problem in Artificial Intelligence. Ablex Publishing Co., Norwood, New Jersey*, pages 99–133, 1996.

[25] Edwin PD Pednault. Generalizing nonlinear planning to handle complex goals and actions with context-dependent effects. In *IJCAI*, pages 240–245. Citeseer, 1991.

[26] Edwin PD Pednault. Formulating multiagent, dynamic-world problems in the classical planning framework. In *Reasoning about actions & plans*, pages 47–82. Elsevier, 1987.

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

[28] Stuart J. Russel and Peter Norvig. *Artificial Intelligence A Modern Approach*. Pearson, 2010.

[29] Blai Bonnet and Héctor Geffner. HSP: Heuristic search planner, 1998.

[30] Wei Zeng and Richard L Church. Finding shortest paths on real road networks: the case for a. *International journal of geographical information science*, 23(4):531–543, 2009.

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

[32] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[33] Richard E Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[34] Eugene F Krause. *Taxicab geometry: An adventure in non-Euclidean geometry*. Courier Corporation, 1975.

[35] AI-Planning-Solver-Shakeys-World-PDDL. `https://github.com/guillaume-chevalier/AI-Planning-Solver-Shakeys-World-PDDL`. [Accessed: 10-April-2018].

[36] International Conference on Automated Planning and Scheduling (ICAPS). `http://www.icaps-conference.org/index.php/Main/HomePage`. [Accessed: 10-April-2018].

[37] Tiago de Paula Peixoto. Graph-tool: Efficient network analysis. `https://graph-tool.skewed.de/`, 2017. [Accessed: 17-April-2018].

[38] KUKA. Kuka youbot. `http://www.youbot-store.com/`, 2018. [Accessed: 7-May-2018].

[39] Miodrag Petković. *Famous puzzles of great mathematicians*. American Mathematical Soc., 2009.

[40] ROS. ROS. `http://www.ros.org/`. [Accessed: 13-December-2017].

[41] ROS. Ros topic. `http://wiki.ros.org/Topics`, 2018. [Accessed: 7-May-2018].

[42] ROS. Ros node. `http://wiki.ros.org/Nodes`, 2018. [Accessed: 7-May-2018].

[43] Sven Koenig, Maxim Likhachev, Yaxin Liu, and David Furcy. Incremental heuristic search in ai. *AI Magazine*, 25(2):99, 2004.

[44] Jörg Hoffmann. FF: The fast-forward planning system. *AI magazine*, 22(3):57, 2001.

[45] Henry Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *IJCAI*, volume 99, pages 318–325, 1999.

[46] Gerald Jay Sussman. *A computer model of skill acquisition*. Elsevier Science Inc., 1975.

[47] Maxim Likhachev, David I Ferguson, Geoffrey J Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In *ICAPS*, pages 262–271, 2005.

[48] Thomas L Dean and Mark S Boddy. An analysis of time-dependent planning. In *AAAI*, volume 88, pages 49–54, 1988.

[49] Rong Zhou and Eric A Hansen. Multiple Sequence Alignment Using Anytime A*. In *AAAI/IAAI*, pages 975–977, 2002.

[50] M Likhachev, G Gordon, and S Thrun. Advances in Neural Information Processing Systems ARA*: Anytime A* with provable bounds on sub-optimality, 2003.

[51] Shlomo Zilberstein and Stuart Russell. Approximate reasoning using anytime algorithms. In *Imprecise and Approximate Computation*, pages 43–62. Springer, 1995.

[52] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Autonomous robot vehicles*, pages 396–404. Springer, 1986.

[53] Johann Borenstein and Yoram Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE transactions on robotics and automation*, 7(3):278–288, 1991.