UiO : **Department of Informatics**
University of Oslo

Reinforcement Learning as a Decision Making Strategy for a Mobile Robot

Bendik Kvamstad

Master's Thesis Autumn 2015

Reinforcement Learning as a Decision Making Strategy for a Mobile Robot

Bendik Kvamstad

3rd August 2015

Abstract

While autonomous mobile robots used to be built for domain specific tasks in factories or similar safe environments, we are now seeing a shift towards the general market. Automated lawn movers and cleaning robots are sold at general stores. They will have to be able to adapt to unknown environments while being safe around humans and animals. This means that we will have to think differently when building the decision making systems for these robots. Reinforcement learning is a field in robotics inspired by humans' ability to learn by trial-and-error. Agents trained with reinforcement learning has been developed and successfully applied to computer games, performing at a world class level.

This master thesis describes the implementation of an AI designed for a robot competing in the 2015 Eurobot-competition. The task is set to a dynamic environment with a robotic opponent. A Goal Oriented Action Planner was implemented as the planner for the AI. In order to adapt the planner to a changing environment, a decision making policy trained with reinforcement learning was utilized to rate actions depending on the state of the game. An implementation of SARSA with feature value function approximation was used to train the policy.

The learned decision making policy showed promising results in experiments conducted for this thesis. The AI found a good static solution to the Eurobot task, and was able to adapt the strategy to a dynamic environment by avoiding the opponent and respecting time limits.

Acknowledgment

I would like to sincerely thank my supervisiors, Associate professor Kyrre Harald Glette and PhD candidate Kim Mathiassen, for their support during the work on this thesis.

I want to thank my fellow students from Robin, FUI and FIFI for all the memorable moments at IFI.

I would also like to thank my friends from TacoTirsdag for making my student years an enjoyable experience.

Last, but not least, I would like to thank my family and my significant other, Emmy Henriette Netka, for continued support throughout the whole process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's world autonomous and mobile robots are entering the general market. While autonomous mobile robots used to be built for domain specific tasks in factories or similar safe environments, we are now seeing a shift towards targeting the general market. Automated lawn movers and cleaning robots are sold at general stores. This means that we will have to think differently when building the decision making systems for these robots. They will have to be able to adapt to unknown environments while being safe around humans and animals.

In this thesis i have looked at how we can build an autonomous mobile robot with multiple tools that can adapt to a new environment. The purpose of the robot is to optimize its performance of goal driven action sequences by learning the environment it operates on. A similar example could be a cleaning robot working in an office environment. The robot is equipped to perform multiple cleaning jobs and should be able to optimize for any given build configuration and job requirement within its domain. Typical examples could be to avoid the hallways during peak office hours or finding a optimal path to collect trashcans.

These tasks may sound trivial, but optimizing them will be essential in the future of robot development when robots will work in environments usually operated by humans.

1.1 Background and Motivation

1.1.1 The Eurobot competition

Eurobot is an international amateur robotics competition aimed at student projects hosted annually around Europe. In this contest the goal is to build a robot that can perform goal scoring tasks on a small playing field while competing against other robots. While the tasks and the outline of the playing field is changed every year, the goal is always to collect more points than your opponent. The main tasks are always in the context of common autonomous robotic challenges like image recognition, position control, avoiding obstacles and collecting objects. This year the competition is hosted in Yverdon-les-Bains (Switzerland) from the 22th to 24th of May.

The theme is "RoboMovies".

1.1.2 The team

This is the first year that UiO is competing in the Eurobot challenge. The team consists of three master students from the robotics group Robin, department of informatics. The main focus for this team will be to lay a foundation for later participants by building a robot that can pass the qualification to the contest. All team members will participate in the main building process of the robot, and also write individual master thesis specializing on different parts of the project. A brief introduction to the projects of other team members are given below.

Andre Kramer Orten will work on the localization and positioning of the robot. For global positioning a IR-system consisting of a tower mounted on the robot and 3 beacon towers around the playing field will be used. Distance sensors will be used for detecting the opponents robot. The master thesis can be found here [28].

Eivind Wikheim will work on the main motor controller system and other actuators necessary for completing each specific task. The design of the robot will be explained more thoroughly later in this paper.

1.1.3 This year's task

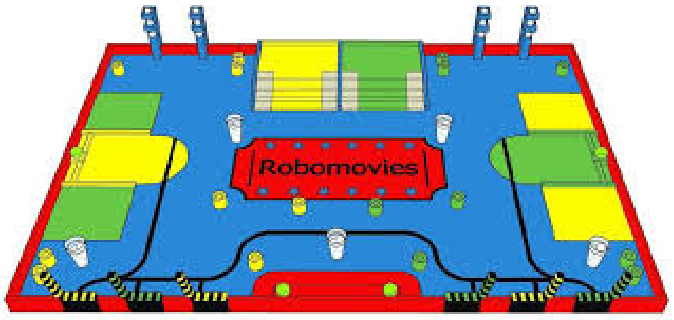

Figure 1.1: Eurobot 2015 playing field

The team RoboMovies takes it inspiration from Hollywood and movie theatres. The tasks include gathering popcorn, building spotlights and clapping clapboards. One of the more difficult tasks is to climb a 10 cm heigh stairway while laying out carpets on each side of the robot.
Tasks Boxes are represented by yellow and green cylinders. To score points the robot must collect boxes and bring them to either its home field or a common delivery area marked as red in the center-side off the map. Extra points are given if the cylinders are stacked in height. Additional extra points can be gained by adding a tennis ball to the top of a stack. A stack of multiple boxes are referred to as spotlights.

Popcorn is represented by white ping pong balls and are located in cups and in popcorn machines placed on the side. To score points the popcorn must be gathered in either cups and placed on the same fields as spotlights or in a special popcorn container in the home base. The popcorn cups are neutral and can be collected and stolen by both teams. Three clapper boards for each team are positioned on the sides of the map. You score points by folding the clapper.
Each team has a stair located on the north side of the map. You score points by climbing the stairs. Additional points can be collected by covering the grey fields on both side of the stairs with a red carpet. The carpet can be preloaded on the robot before the game starts. The team has decided that a second smaller robot will be required to perform the stairway climbing, and I will not focus on handling the stairway in this thesis.

1.2 Problem definition

The goal of this project is to implement an AI for a mobile robot that utilizes Reinforcement Learning as a decision making strategy. The AI will be used for a robot competing in the Eurobot 2015 competition. A major part of the project will also consist of building the actual robot. The AI must be able to plan a strategy for a match and tackle hindrances during the game. Operating in a changing environment, the robot must be able to recalculate plans fast and efficiently in order to avoid collisions and loss of precious time.

The A.I should fulfill the following requirements:

- The AI should be able to calculate safe and efficient plans for collecting game objects and scoring points.
- The planning system must be able to efficiently recalculate new plans to handle changes to the playing field during a match.
- The robot is not likely to be finished until a few weeks before the competition. In order to test the AI thoroughly, a game simulator must be built. The simulator should run in real-time displaying the robot, the game objects and the opponent robot. It is also preferable that the simulator can display real data from the robot when the construction is complete.

1.3 Outline of the thesis

The thesis consists if six chapters: introduction, background, robot, implementation, experiments and results, and discussion.

Chapter 2 contains general background information about the theory which this work is based on, including an overview of Planning strategies, Reinforcement Learning, Pathfinding and previous work this thesis is built on. Chapter 3 contains details on the robot that was built for the Eurobot competition. Chapter 4 describes the implementation of the AI and the decision making policy based on Reinforcement Learning. Chapter 5 then outlines the experiments that were conducted, and presents results as well as an analysis for each experiment. Chapter 6 contains a general discussion and a conclusion.

Chapter 2

Background

Humans and human behaviour has always been a source of inspiration in the field of robotics. Robot learning is a fairly new research field at the intersection of machine learning and robot design. Learning algorithms allows a robot to discover optimal behaviour through trial-and-error interactions with its environment.

2.1 Planning

A fundamental problem in robotics is to construct systems that can translate high-level specifications of humanly tasks into low-level orders of robotic movements. In Artificial Intelligence, an agent can use planning algorithms to construct orders of actions to perform in its environment. However, solving tasks in complex dynamic enviroments is a difficult problem [2], and planning in robotics has evolved from simple motion control to include decision making and game theories. More information on the background of planning algorithms can be found here [17] [3]

2.1.1 Definition

Planning is a branch in Artificial Intelligence that concerns decision making and the realization of strategies for intelligent agents or autonomous robots [20]. To reason about what to do, an agent must have goals, a world model and a understanding about the consequences of its actions.

A general planning problem consist of:

- A initial world state S
- A set of actions descriptions A
- A goal state description G

The planners objective is to find a sequence of actions, or plan, such that executing them in the initial state will change the world to satisfy the goal state description.

2.1.2 Terms

State: In a planning problem, the state-space represents all possible outcomes [17]. A state can for example be a specific position in a chess game and the state-space would contain all possible moves. A state can also contain information about the robot, for example the position of a robot.

Time: All planning algorithms must consider the concept of time. It is not limited to the minutes and seconds of a clock, but could also represent distance travelled or KWs used.

Actions are the tools that robot can use to interact with the environment and change its state. Motions can vary from movement to recognizing and picking objects.

Plan: A plan may simply be a sequence of actions an agent must perform to achieve a goal. However, a plan can be dynamic and react differently to future states for example in a non deterministic environment. In this example the action is a function of state.

Initial and goal state: Initial state is considered as the starting position when a new plan is calculated. A goal state is a state where the agent has completed a specific task. A planner can have multiple initial state and goal states. In a game of chess the initial state can be the board configuration of a new game, and a goal state can be any state where a winner is declared or a draw is agreed.

2.1.3 FSM

A Finite state-machine (FSM) is a model used to represent and control behaviour and execution flow [8]. In a FSM the behaviour of an agent is represented as a finite number of states. A simple example of a FSM model can be seen in figure 2.1. The agent can transition between states, but can only be in one state at a time. The state transition can be triggered by an input event from sensors or an internal condition like a timer. An agent working in a complex state-space can structure the control logic into a hierarchie of FSMs. A typical solution would be to separate movement control and action selection into separate FSMs and create a higher order controller that decides which FSM is active [8]. More information about FSMs can be found here [18] [11].

FSMs are hard to generalize and are typically hand coded for specific problems [25]. This result in a heavily customized planning algorithm which can be hard to maintain and update for new features.

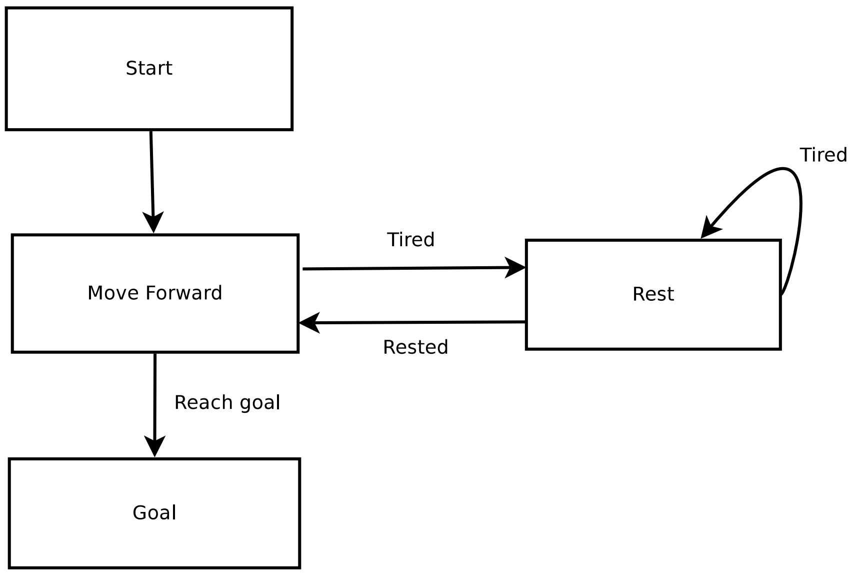

Figure 2.1: Model of a Finite-state-machine

2.1.4 Markov Decision Processes

A Markov decision process (MDP) is a probabilistic decision making framework that can be used to model planning problems [13]. In an ideal world from a planner perspective, all environments would be deterministic and fully observable. Unfortunately, this is often not the case. When an agent performs an action in a dynamic environment, there is a possibility of failure. MDP can be explained as a process that has a state s and a set of possible actions A. A decision maker picks an action a from A, and the process responds with a new state and an accompanying reward based on the performance of action a in s.

The core problem for an MDP is to find a policy for the decision maker. A policy can be defined as a function π that specifies the action $a = \pi(s)$ that the decision maker will take when in state s. The goal of a MDP-planner is to find a policy that maximizes the rewards returned from performing $\pi(s)$ for all states. A extensive research of different policy strategies can be found in this article [19]. Reinforcement learning is one of the more popular policy strategies developed over the last years and will be discussed in section 2.2.

2.1.5 GOAP

Goal Oriented Action planning is an AI system that allows an agent to plan a sequence of actions to satisfy a specific goal. The planning algorithm was first introduced in [27]. The plan does not only depend on the particular goal but also the state of the environment and the internal state of the agent. A simple example can be seen in figure 2.2. In order to satisfy a goal, the agent has to search for actions that will forfill the requirements for this state. All actions in GOAP has preconditions, effects and costs. By using the preconditions and effects, the algorithm creates a graph of reachable states and the actions that are required to achieve the specific states. Using a graph search the agent can find the sequence of actions that are required to reach its current goal state.

Decoupling states and actions in such a manner makes the GOAP algorithm very flexible and easy to maintain for agents with large state-spaces. And using preconditions and effects, there can never exists a plan with illegal sequences of actions. Using graph search algorithms like A* and the cost of actions, the algorithm can be optimized to always choose the lowest cost plan to achieve a goal.

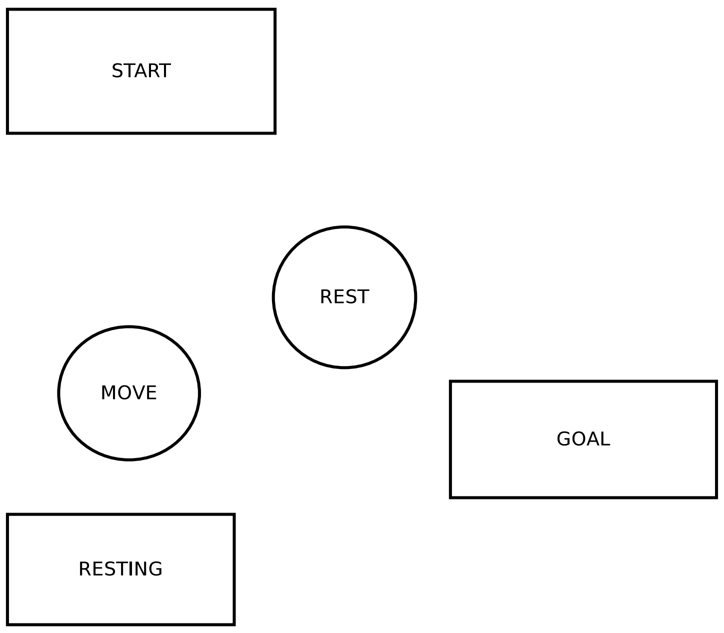

Figure 2.2: GOAP model

2.2 Reinforcement learning

Reinforcement learning (RL) is the name given to a range of techniques for learning based on experiences. RL bears striking resemblance to the learning process of humans and animals: actions that are rewarded tend to occur more frequently; actions that are punished are less likely to be repeated [32]. The first remarkable breakthrough in reinforcement learning was the TD-Gammon, a backagammon AI created by Gerry Tesauro in 1992 [33]. TD-Gammon required little backgammon knowledge, yet learned to play near the level of the worlds strongest grandmasters. In later years, agents trained with reinforcement learning has been developed and successfully applied to games, performing at a world class level [24].

2.2.1 Definition

A reinforcement algorithm in its most general form has three components. An exploration strategy for trying out different actions, a reinforcement function that evaluates the performance of an action, and a learning rule that links the two together.

A standard reinforcement learning model consists of

- A set of states S
- A set of actions A
- Rules that determine the reward of performing action a in state s
- Rules that describe what the agent observes

A reinforcement learning agent interacts with an environment E in a sequence of actions, observations and rewards. At each time-step the agent selects an legal action a from A. The action is then executed on the environment and updates the state. The learning agent then observes the new state s' and a reward r representing the performance of action a in s. The agents learning task is to execute actions in environment, observe results and learn a policy $a = \pi(s)$ that maximizes the future reward. A detailed introduction to reinforcement learning can be found here [14]. RL implementations for mobile robots are discussed in [30] and [21]

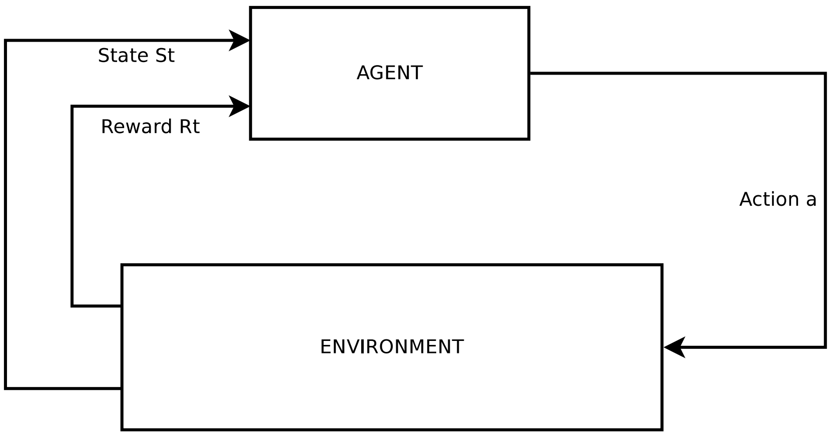

Figure 2.3: Reinforcement Learning model

2.2.2 Terms

Unsupervised learning
The area of machine learning where an agent learns by interacting with its environment rather than from an experienced teacher.

Policy
A policy determines how a learning agent should behave at any given time.

The policy is typically formed during a learning phase by experimenting on a environment.

Value function
A mapping from states to real numbers, where the value of a state represents the long term reward for applying a particular policy.

2.2.3 Q-learning

Q-learning is a model-free reinforcement algorithm named for the set of quality information (Q-values) it holds about each possible state and action [36]. The algorithm treats the environment as a state machine. At any point in time the algorithm is in some state. The state should represent both internal information about the agent and external knowledge about the environment. For example the battery level, tasks completed, agent position, distance to obstacles. Any information that is not in the state cannot be learned [29]. The Q-learning algorithm learns by testing out every possible action for every state, and evaluating the performance. Over time the algorithm will learn which action that performs best for each state at a given time. More information on GOAP can be found here [23] [29].

The algorithm keeps track of a history of experience tuples, often written as $< s, a, r, s' >$, where s is the start state, a is the action performed, r is the reward and s' is the resulting state. The experience tuple is divided into two sections. The first two elements $< s, a >$ is used to look up the Q value in the algorithms memory. The second two elements $< r, s' >$ are used to update the Q-value based on the reward of the action and how good it will perform in the next state s'.

The update is handled by the Q-learning rule:

$$Q(s,a) = (1-a)Q(s,a) + a(r + ymax(Q(s',a')) \tag{2.1}$$

where a is the learning rate and y is the discount rate.

The first component $(1-a)Q(s,a)$ is just the current Q-value for the state and action. Using this information means that we never lose previously discovered experiences. The second component is the reward of the s, a and the maximum potential of the future state s', a'. So in addition to the reward of doing action a, we also consider the Q-value of all possible actions that can be taken from the new state s′. This helps rewarding actions that has a high future reward. For example in Backgammon where the only rewards are Loose -100 and Win +100, the reward from the finishing move will trigger down to the next to last finishing move, then to the next to last finishing move, and so forth.

2.2.4 SARSA

SARSA is an On-Policy variation of the Q-learning algorithm. The difference between SARSA and Q-learning, is that the maximum reward for the next state is not always used for the updating the Q-values. Instead of selecting the max-value, SARSA chooses a new action a' for state s' using the same policy as for s, a. The name SARSA comes from the representation of the experience quintuple $< s, a, r, s', a' >$. In simpler words, SARSA chooses the next state based on what it expects to be the best move, given its current policy. Q-learning chooses the next state based on what could be the best move, if the current policy is perfect. More information on SARSA can be found here [31].

SARSAs Q-learning rule:

$$Q(s,a) = (1-a)Q(s,a) + a(r + y(Q(s',a'))) \tag{2.2}$$

As described above, s' and a' is selected with the same policy as s and a.

2.2.5 Value function approximation

A big challenge with reinforcement learning is handling the amount of states [22]. A problem like Backgammon has so many possible states that it would not be feasible for a reinforcement learning algorithm to evaluate every state and action. The alternative to reasoning explicitly in terms of states is to reason in terms of features. With feature value function approximation we look to train a value function which can calculate the best action for each state. A set of weights W which will indicate which action is best for state s.

Features are functions of from state-action pairs to real numeric values that capture important properties about the state. Example of features could be the distance from obstacles or a distance to objectives. Using a feature representation we can write a value function (Q-function) for any state using a few weights.

$$Qw(s,a) = w0 + w1F1(s,a) + ... + wnFn(s,a) \tag{2.3}$$

Advantages are that each state-action pair can be summed up in a simple value.

2.3 Pathfinding

In its most basic form, path planning in robotics is about finding a route from a to b while avoiding collisions. Pathfinding will not be the main focus of this thesis, however a good algorithm fitted for the task will be vital for the project to work. In order to find a suiting algorithm, state of the art path finding algorithms will be researched.

2.3.1 Theta*

Due to its simplicity, A* is the most common algorithm used for graph traversial of a roadmap. This is because the algorithm is guaranteed to find a shortest path on the graph. The problem with A* and many other graph traversial algorithms is that a shortest path on the graph is not equivalent to a shortest path in the continous environment. The algorithm is limited by the propagated information in the graph. Theta* builds upon A* first presented by Koenig in this article [26]. The main difference between Theta* and A* is that the former allows the parent of a vertex to be any vertex in the graph. By allowing any-angle paths, Theta* can heavily optimize a path found by A*. Considering a A*-path from $(qi, qa), (qa, qb)$. If there exists a line of sight from qi to qb without obstacles, qa can be dropped from the path. While Theta* is not guaranteed to find shortest paths in the continous environment, it does find the shortest paths a large percentage of the time.

2.3.2 Potential fields

Potential field methods (PFM) falls under local path planning methods and the idea behind it is taken from nature. Consider a small ball rolling down a hill. The idea is that depending on the slope of the hill, the ball will always move down into the valley. In robotics we can simulate the same effect by creating artificial potential field that will attract the robot to the goal.

In a simple environment without obstacles we would only need to create a single attractive potential field inside the goal position qg [12]. The potential field is defined across the entire cfree, and over time calculating the force applied to the robot, the robot will move against the goal. To avoid obstacles, we place negative potential fields over every obstacle. While being forced against the goal, the robot will simultanously be pushed around obstacles. In mathematical terms, the overall potential field is:

$$U(q) = Ugoal(q) + SUM(obstacles(q)) \tag{2.4}$$

and the induced force is:

$$F = -U(q) = (U/x, U/y) \tag{2.5}$$

PFMs were originally used for collision avoidance and has a problem with being stuck in local minima when used for global path planning [38]. An example is a robot being trapped in a U-shaped obstacle. Several methods has been suggested to deal with this problem. One of the most common solutions is to use PFM in combination with a global planner. Another is to let the robot get trapped in a local minima and use backtracking or random movements to help it escape. All this approaches are based on that the robot can discover that its trapped.

2.3.3 Probabilistic roadmap

Probabilistic roadmap methods (hereby noted as PRM-) work in two phases. A learning phase and a query phase. In the learning phase, the algorithm constructs a map over all collision-free configurations in the configuration space. These configurations form the vertices in a graph called a roadmap. A simple local planner is used to look for connections between connected vertices. If a connection is found, an edge is added to the graph. On completion the roadmap will give a sufficient representation of the collision-free map.

In the query phase, the initial configuration qi and the goal configuration qg are connected to the graph using the same local planner as was used in the learning phase. Finding a path from qi to qg is reduced to a simple graph search in the roadmap.

The learning phase of PRM is the cost heavy operation, and must be taken into consideration when deciding to use this algorithm. For robots working in a static environment, the roadmap can be preloaded and efficiently calculate paths.

2.4 Related Problems

In order to find previous related work on the type of algorithm that was to be implemented, it was necessary to find similar problems that have been successfully solved with learning strategies. In this section, two well known algorithms are described, the Traveling Salesman Problem and Vehicle Routing Problem. None of them matches the Eurobot task completely, however ideas and previously used techniques might be useful for the implementation of the algorithm discussed in this thesis.

2.4.1 Traveling Salesman Problem

The traveling salesman problem (TSP) is one of the most researched and famous algorithmic problems to date. A travelling salesman has to travel around to nearby cities in order to sell cargo from his car. Being a businessman, the salesman doesn't want to waste time on the road and therefore want to find the most time efficient route that allows him to visit all the nearby cities. So the salesman asks for the shortest route to visit a collection of cities and return to the starting point.

Definition
Given a set of towns and the distances between them, determine the shortest path starting from a given town, passing through all the other towns and returning to the first town.

Common solutions A general overview of solutions to the TSP problem can be found here [15] Below i will give a brief overview of some of the more common learning strategies used for solving TSP.

Genetic Algorithms
Genetic algorithms(GA) is an evolutionary algorithm inspired by biological evolution. GAs uses the principles of nature selection to solve complex algorithmic problems. Starting with a population of proposed solutions for the problem, the GA uses mutation and crossover between individuals to improve their fitness. The algorithm continues this evolution until a terminal condition is met. A detailed introduction to GA is given in [1]. [9] gives a good overview of different GA approaches for solving TSP.

Ant Colony Optimization (ACO)
Ant Colony Optimization is a heuristic search method for solving difficult problems in combinatorial optimization. The method is inspired by the biological behaviour of real ant colonies. When ants try to find the shortest path between their nest and a food source, they drop pheromones on the ground creating a track. Over time the pheromone will evaporate, and the more time it takes the ant to walk the track, the more the pheromone will evaporate. However a shorter track will be walked faster and the pheromone will decay slower. Other ants will tend to follow tracks with strong pheromone, and therefore prefer the shortest path. The idea of ACO is to mimic this behaviour with simulated ants walking around the graph. More information about solving TSP with ACO can be found here [37].

Q-Learning
Some recent studies have been done using Q-Learning approaches to solve TSP. An overview of Q-learning was given in section 2.2.3. A variation of ACO with Q-learing has shown promising results [4]. In [7] an implementation of SARSA for solving TSP is proposed.

Relevance The Eurobot task of collecting objects has similarities to the TSP, however the task is more complexe. In the Eurobot task we also have to consider the capacity of the robot.

2.4.2 Vehicle Routing Problem

The vehicle routing problem (VRP) is a combinatorial optimization problem seeking to service a number of customers with one or multiple vehicles. VHP is a generalization of TSP. A thorough review of VHP can be found here [34].

Definition
The objective is to optimize the path for a delivery truck that has to visit a certain number of customers in a city. The vehicle has a limited capacity and will need to return to one or multiple depots to offload goods.

A general review of algorithm for solving VRP can be found here [16]. The learning based approaches are usually variants of the approaches explained under the previous section where capacities are added to the problem.

Relevance The vehicle routing problem is very similar to the Eurobot task and work done on learning approaches will be researched for this thesis.

2.5 Previous Work

This section presents some of the previous work done on learning strategies for problems similar to the Eurobot task. Implemented systems will be briefly discussed, along with thoughts on how the can benefit this project.

Training Pacman with advanced reinforcement learning

In [35] the author presents an agent traind with reinforcement learning that learns to play the classic arcade game Ms. Pacman.

Overview:
In Ms Pacman, the goal of the agent is to eat all the food on a map in a maze like environment, while avoiding ghosts. In the article the author implements a SARSA reinforcement learning algorithm that will train pacman by simulation. This implementation differs from regular grid-based RL implementations in the representation of states. A common approach is to see every grid as a state, and calculate which grid to move to next depending on threat and value of the grid. Here, the author has chosen states represented as a 10-dimensional feature-vector. The features holds information about the adjacent grids and threat indicators. With a complete state-space $4 * 2^9 = 2048$, the problem can be solved with on-policy learning.
The implementation shows good results in simulation, and the agent is able to learn a good policy that can be used on different environments.

Usefulness:
The Ms. Pacman environment is quite different to the Eurobot problem, however there are some similarities that are worth noting. The agent was able to learn how to avoid ghosts, by preferring actions in areas where there are no threats. This indicates that a similar solution could work for the Eurobot agent, preferring tasks away from the opponent robot. However, the Eurobot agent and environment is more complex than the envorionment discussed in this article. The Eurobot agent has free motion, while pacman is limited to four different moves. Finding a small enough state representation that allows an exact value function for the Eurobot task is not very likely. An approximated value function will allow for a more detailed state representation and a larger state-space, however the performance is not guaranteed to be as good.

Hybrid genetic algorithm solving Vehicle routing problem

This master thesis [5] presents an implementation of a dynamic schedulerer solving a variation of VRP using genetic algorithms.

Overview: This article is based on a previous Eurobot task from 2010. The task has many similarities to this years task, and the problem can be reduced to a dynamic vehicle routing problem. A planner was implemented for a harvesting robot, gathering boxes from a playing field and delivering them to a depot. The robot had a limited capacity and had to return to the depot multiple times. The author describes a hybrid genetic algorithm that was optimized as a schedule generator for the robot.

Usefulness:
The proposed genetic algorithm showed good result in finding a static solution for a problem very similar to this years Eurobot task. The thesis also describes difficulties involved with building a robot for the Eurobot competition. It highlights the importance of a simulator.

Playing Atari with Deep Reinforcement Learning

This article [24] presents an agent trained with Deep Reinforcement Learning that plays classical arcade games at a world class level. Overview: [34] demonstrates that a convolutional neural netowrk can learn control policies from raw video data in complex RL environments. The network is trained with an implementation of the Q-learning algorithm, with stochastic gradient descent to update weights. Using a Atari console simulator, the RL-agent is trained using a video stream as input and a reward function directly from the games. The agent was not fed any information about the rules or concepts of the games. Using different image transformation algorithms as input, the agent was able to learn and play the games at a world class level. The algorithm outperformed all previous approaches on six of the games and outperformed a human expert on three of the thirty seven games.

Usefulness: The proposed algorithm in this article has shown great results, and has been a inspiration for selecting reinforcement learning strategies for this thesis. However, the approach taken in this article requires complex image recognition algorithms and might not be suitable for this project.

2.6 Discussion

There were two main considerations that had to be taken when selecting an implementation for this thesis. An AI had to be implemented for the Eurobot competition, and also allow for testing with reinforcement learning. A Goal Oriented Action Planner was chosen as the planning algorithm for the robot. The GOAP-planner was flexible enough so that a trained decision making policy could be incorporated. If the robot was completed early, the decision making policy could easily be replaced with a static solution for testing. SARSA was chosen as the reinforcement learning algorithm in combination with a value function approximation strategy, due to the complexity of the state-space in the Eurobot task. The GOAP-planner will use the value function of the learned policy when comparing the value of actions while creating plans.

Chapter 3

Robot

Building a robot for the Eurobot competition is a main part of this master project. This section is intended to give the reader a better understanding of the robot the AI was implemented for.

3.1 Design

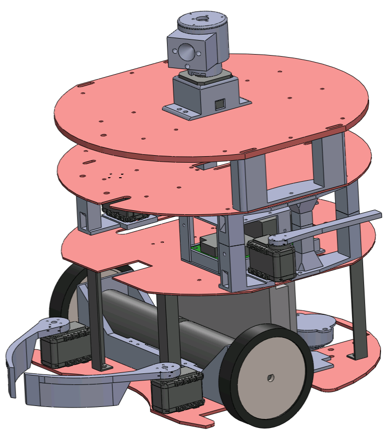

Figure 3.1: A model of the robot

The design and building of the robot was completed a few weeks before the competition. Running on a low budget, the robot was built by using a robot kit as base layer. The parts of the kit are the orange plates on figure 3.3. The other parts were 3d-printed at the university lab. The size of the robot was mainly decided by the Eurobot rules [6].

The robot is 30 centimeters high and has an oval shape. The IR-tower module is located on the top of the robot. Using Infrared the tower communicates with 3 beacons around the playing field. With triangulation the module can keep track of the robots position. The distance sensors are also placed on the top level. A ultrasound sensor is used in the front where a greater distance is needed. Two infrared distance sensors are located on

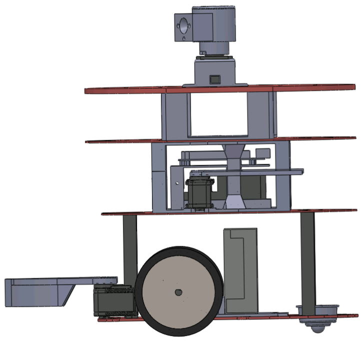

Figure 3.2: The robot seen from an angle.

the back.

One arm on each side of the robot is used for flipping the flip-boards. From here on the arms will be referenced to as shutter left and right. The two grippers in front will be used for collecting stands and popcorn glasses.

The third level contains a laptop and two arduinos. One arduino drives the IR-tower and the other drives the distance sensors. The laptop will run the AI, motor controller and the navigation system. The different software modules will communicate asynchronously through sockets.

3.2 Modules

Being a team consisting of 3 members working on the project, we decided to modelize the robot. The robot was separated into 4 indiviudal models, Motor Controller, Localization, Vision and the A.I. The modules were built as indiviudal systems and communicated through socket messaging on the computer located on the robot. The A.I is discussed further in section A.I. Below i will give a brief introduction to the other modules.

3.2.1 MotorController

The MotorController module is the controller system that handles actuators and the movement for robot. The MotorController has an API which allows the AI to queue commands. States of the different motors can also be retrieved from the API.

Driving

The motor controller has an internal grid system of the Eurobot playing field. In order to move the robot around the map the AI requests a move with $Position(x,y)$ to the MotorController API. During a driving phase, the AI can poll updated information about the current position and driving state. The MotorController does not handle obstacle avoidance or any other game based state. It is the AIs job to find a path around obstacles and queue positions that forfill such a path.

Actuators
The gripper and flippers seen in figure 3.5 are used for performing actions to score points in the competition. As with the driving system, the actuators can be steered directly through the API by sending a request for a transition to a new angle.

3.2.2 Localization

A rotating IR-tower is located at the top of the robot, as seen in figure 3.5. A global positioning system is built around the system. By using three beacons located on the side of the playing field, the IR-tower uses triangulation in order to determin the global position of the system.

3.2.3 Vision

Unfortunetaly we were not able to build a functional vision system for the robot, and had to fall back on distance sensors located on the top of the robot. Without a global positioning system, the AI could not determine the position of the opponent.

3.3 Module communication

Building a complete autonomous robot requires a complex system of controllers, sensors and actuators. For the robot discussed in this thesis, we decided to seperate the system into individual modules. As mentioned in 3.2 the A.I and MotorCotroller will run on a intel computer with linux. All the modules will run on separate processes. For communication between the modules, a socket library named zeroMQ (zmq) was chosen.

3.3.1 ZeroMQ

ZeroMQ is a high-performance asynchronous messaging library. Using sockets the library allows for fast communication between applications both locally and over network. A typical setup is that one application acts as a threaded server and the other application connect as a client. ZMQ is cross platform and available for multiple languages. More information about ZMQ can be found here [10]

In our project we have multiple applications written in different programming languages. Using zmq we can send messages almost instantly between the A.I written in Java and the MotorController written in C++.

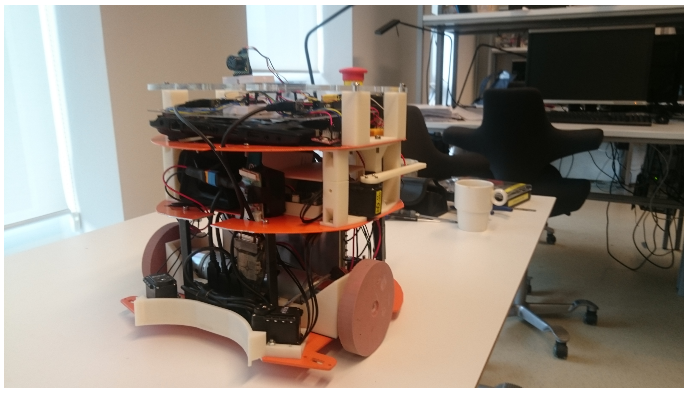
Figure 3.3: The robot built for the Eurobot competition.

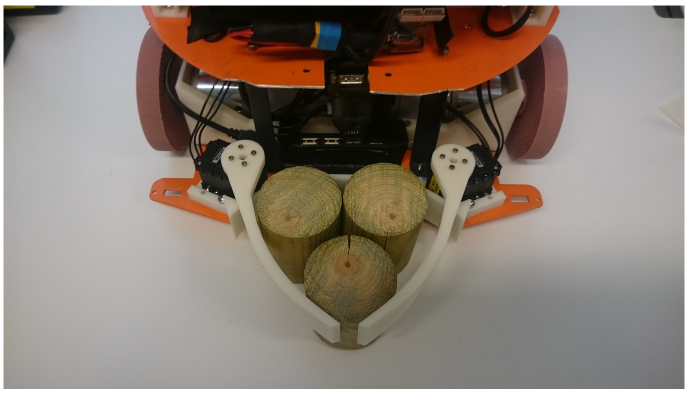
Figure 3.4: The robot's grippers, used for gathering boxes.

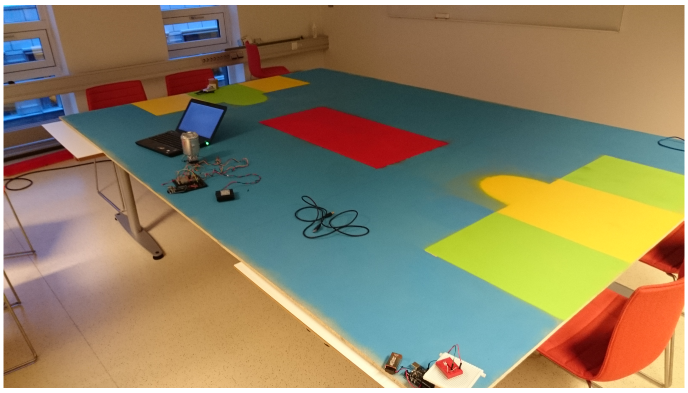
Figure 3.5: A replica of the original Eurobot playing field was built for testing.

Chapter 4

Implementation

4.1 AI

The main controller system for the robot described in chapter 3 will be referred to as the AI. The AI can be seen as the brain of the robot. Its job is to plan, calculate and perform actions for a robot competing in the Eurobot competition.

4.1.1 Controller system

The top level controller system was implemented as a finite-state-machine shown in figure 4.1. A control loop updates the FSM in 60ms intervals. Below is a description of the different states.

Ready
Following the Eurobot rules [6] the robot must be started manually with a start cord. In this state the AI will listen to a trigger on the start sensor. When the thread is pulled the AI will transist to the next state.

Calculate plan
In this state, the AI would request a plan from the GOAP-planner.

Perform plan
In this state the AI would execute actions until the current goal was completed.

Emergency stop
In order to avoid crashes, the AI had an emergency stop system. Using distance sensors, the AI could continously check for threats. If a threat was detected, the robot would shut down, and ask the GOAP-planner for a new updated plan.

Done
A Eurobot match lasts for a total of 90 seconds. If a robot continues to move after the time limit, the team risks disqualification from the contest.

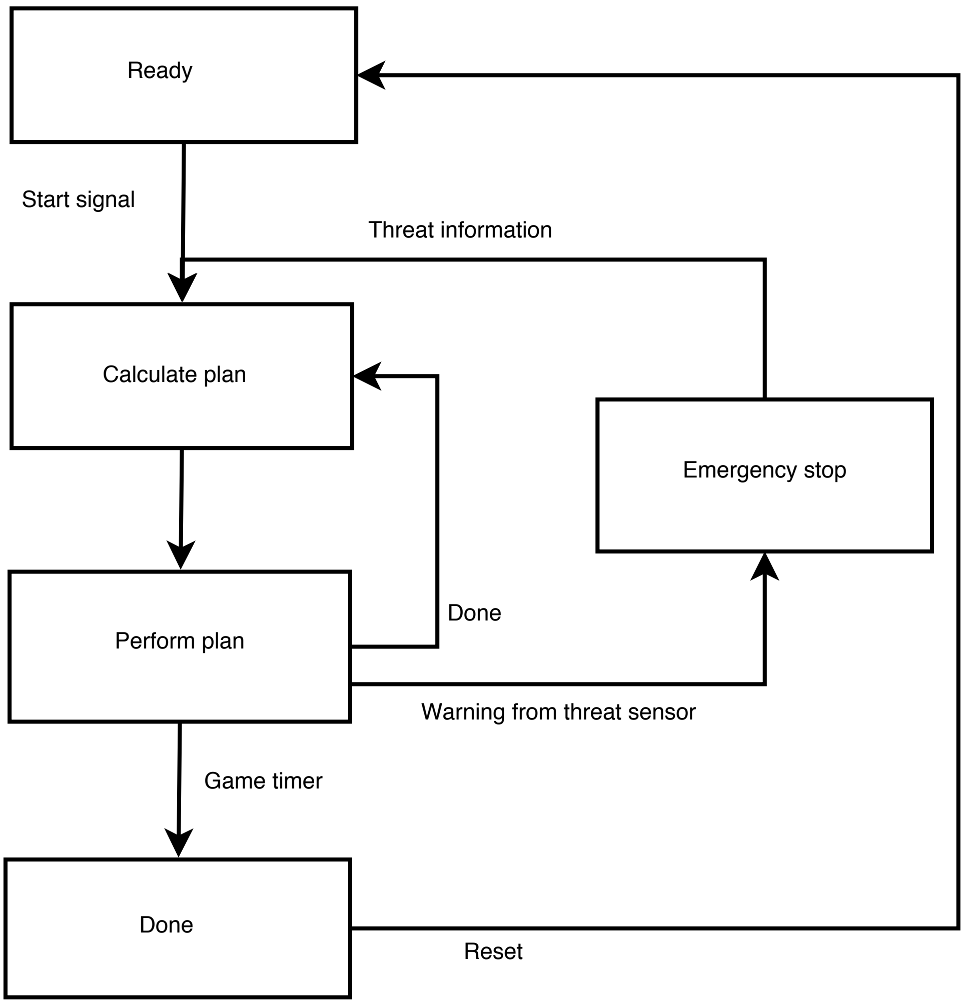

Figure 4.1: Overview of the main controller system AI.

Therefore it is very important that the robot shutdowns in time. A timer is implemented in the AI, and updated every iteration. If the timer reaches the limit, a kill-signal is sent to the driving system and all actuators.

4.1.2 Planner

A Goal Oriented Action Planning algorithm was implemented as the planner, and is described in detail in section 4.2. The planner was used in two of the states of the controller system. During the

4.2 Goal Oriented Action Planning

A implementation of a Goal Oriented Action Planner was implemented as the planner for the AI. The GOAP-planner will us a trained decision making policy in order to evaluate different actions for a given state. This section will give a brief overview of the GOAP implementation.

4.2.1 States

A GOAP state is a collection of state variables derived from the current board state of the game. The state is used when the planner derives available actions and to see which actions can be executed, depending on their precondition. The state must capture all the important infromation from the environment. Most of the variables are the conditions of game objectives and the current status of the robot.

4.2.2 Actions

For this implementation of GOAP it was decided that the responsibility of movement would be removed from the planner and therefore not considered as a GOAP action. The planner should only decide which objective to do next, and the actions themselves will be responsible for the movement while performing the task. However, travel distance and positioning will still be considered when the actions are evaluated. As discussed in section 2.1.5 every goap-action has preconditions and effects.

PickBoxAction (PBA): The action of collecting a box from the game board. Collecting boxes is one of the main objectives of the Eurobot task this year to score points. The PBA will control the whole movement of opening grippers in a feasible distance from the box, positioning the box and closing the grippers. In order to perform this action there must be available boxes on the playing field, and the robot must have the additional capacity

PickPopcornAction (PPA): This action is very similar to the PBA, however the preconditions and effects are replaced with states related to popcorn stands. There is also a small difference in the movement performed while collecting.

DeliverGoodsAction (DGA): In order to score points, all collected objects must be delivered to a depot. The DGA handles the movement and releasing the objects inside a valid depot. In order to perform this action the robot must be carrying cargo and a open depot must be available. Collected cargo that is not delivered to a valid depot before the time limit will not gain the robot any point.

FlipClapperAction (FCA): The action of flipping a movie clapper located on the sides of the board. Two triggers located on each side of the robot will be used to perform the action. The FBA is responsible for the opening and closing of the trigger and positioning correctly next to a move clipper in order for the action to be performed succesfully.

4.2.3 Goals

A goal is a state configuration that satisfies a certain goal predicament. GOAP is a goal driven planning algorithm, and will therefore search the action space for a valid sequence that leads to a goal state. For goals that need a large number of actions, the amount of possible permutations of the action sequence can grow infinitely. It is therefore recommended to use a heurestic to speed up the search. With multiple goals, either a priority rank or a most valued first approach can be taken. Below i will give a short introduction to the goals used for the Eurobot task.

BuildTowerGoal: The build tower goal requires the robot to gather boxes around the playing field and delivering them to a valid depot. This goal can only be reached from the deliverGoodsAction. This goal can be completed multiple times, as long as there are open depots and available boxes on the playing field.

DeliverPopcornGoal: This goal is similar to the buildTowerGoal, but for gathering and delivering popcorn cups.

FlipClipBordGoal: This goal is for flipping the ClipBoards located on the side of the playing field. This goal is only completed when there are no unflipped ClipBoards left.

4.2.4 Creating Plans

As discussed earlier a plan is created by searching for a valid action sequence that leads to a goal. Given the current state and a goal, the algorithm will build a graph over possible actions and state-space. In this implementation A* is used to search the state graph. The algorithm starts off by identifying the available actions in the current state. It then chooses the most valuable action given by a heuristic that will be discussed later in this section. The action is then applied to the state and new state s'. If s' is a valid goal state, we have a new valid plan. If not, s' will be added to a queue of possible states, and traversed for possible actions in s'. This will be done for every action available in the state-space until a valid plan is found, or none exists.

4.2.5 Evaluating Plans

A good heuristic is the foundation for a succesfull A* algorithm. A common heuristic in A* for GOAP is to calculate a distance between current state and goal state. For example if the goal Sg is to collect 5 boxes and you currently in S1 have 2 boxes, you have the distance 3 to the goal state. This means that a state S2 with 3 boxes would be "closer" to the goal state than S1. All though this heuristic might work excellent for simpler problems, it has its faults when the problem becomes more complex. Given the earlier example, if there are more than 5 available boxes, you would like the heuristic to also choose the route with the shortest driving path. For this years Eurobot task we are also working with a changing environment. This means that a plan that was optimal at $t1$, might not be possible to do at $t1 + n$. It would therefore be preferable if the heuristic could also consider the likelihood of future states happening and prioritize safe plans. In order to forfill all these requirements it was decided that an decision making policy trained with reinforcement learning should be used as a heuristic for the planner. This algorithm will be discussed in the next section.

4.3 Value Function Approximation

The decision making policy consisted of an implementation of SARSA with value function approximation. The goal is to find a decision making policy that can help the GOAP-planner choose the best actions at any moment of the game. With the Eurobot task being a dynamic game, the value function would have to consider both the current state of a game and changes that may possibly happen in the feature. In order to achieve such a value function, experiments will be done for different learning strategies and feature extractors. The policy will always choose the action with the highest score given from the Q-value function:

$$Qw(s,a) = w0 + w1F1(s,a) + ... + wnFn(s,a) \tag{4.1}$$

The weights are values learned from training. A high valued weight indicates that the specific feature will be impactful if the feature value also is high. For example if the weight for the feature DistanceFromOpponent is set to 100, the policy will prioritize actions that maximizes the distance. If the weight DistanceToObjective is set to -100, the policy will prioritize actions where the feature value is low.

4.3.1 Weights

In this implementation the weights are represented by a vector of floats. As by the learning rule equation 2.2 the weights will be updated during the learning phase with the goal of forming a good policy. The weights can be seen as the memory or brain of the algorithm. The weights are update by a combination of the reward recivied for performing an action in a state. The planner will choose the action that returns the highest value from a action state pair using the value function.

4.3.2 Features

Feature selection is one of big difficulties with value function approximation. Experiments on different combinations of features was done and can be reviewed in [Experiments features]

In a SARSA implementation each feature is calculated from a state and an action. Therefore it will also hold information on the objective of the action, for example a box or a popcorn cup. Below follows a brief discussion on the features used in this implementation.

Robot position: The robots position is represented in 5 features. 1-4 is distance from north, west, east and south border of the game board. The fifth is the rotation of the robot.

Opponent position: The opponent robots position is only relevant in relative distance to our robot and the distance to objectives. Therefore there is two features f5 distanceToOpponent and f6 distanceToObject. Distance to object will be 0 if we currently do not have a object.

Time: In order to prioritize plans that finish in the time range of a match, features for time is a must. Feature fx is time from beginning in the range [0, 90] and fx is time left [90, 0].

Object value: While optimizing the performance of the static solver and selecting the most valuable route, features with action target specific information was added. This includes for example distance to nearby objects.

4.3.3 Learning parameters

The learning rule described in 4.2 has three parameters that highly effects the performance of the algorithm. A brief introduction to the parameters is given below.

Alpha (a) - The Learning rate The learning rate, a value from [0, 1], controls how much influence the current feedback value has over the previously learned Q-value. A high learning rate would give more credit to new experiences and a low learning rate would value previous learned rules higher. The learning rate can also change over time. In the initial part of the learning process the value can be relatively high since the stored q-values have confidence. As the learning process evolve, we can rely more and more on our past experiences and the learning rate decreases.

Gamma (g) - The discount Rate The discount rate controls how much a <State, Action> pairs Q-value depends on the q-value at the state it leads to. A value of zero would disregard all future stats and only consider the immediate reward of the action. A value of one will value them equally. Higher values favor longer sequences of actions, but requires a longer learning process. In the Eurobot task, we have multiple actions that are necessary but redeem no immediate reward, like collecting boxes or popcorn cups. Therefore a gamma > 0 is required.

Rho (r.) - Randomness for Exploration All reinforcement learning algorithms require a degree of randomness in order to learn. Without it you would constantly be stuck in local maximums, and learning would not be possible. Rho controls how often the algorithm will take a random action, rather than the best action based on previous learning. Rho is a value from [0, 1].

The learning parameters for this implementation will have to be found by testing, as the optimal parameters are problem specific.

4.4 Simulator

As the goal of the project stated, a simulator had to be implemented in order to test the A.I. while the robot was under construction. A simulator that could simulate a complete Eurobot game was built from scratch. In addition to a score and fitness evaluator, a 2d-graphical visualization of the simulator was also implemented. The Eurobot Simulator (EuroSim) was constructed with two different modes. A simulation mode where the motor controller and all sensors where simulated, and a Live-mode where the EuroSim used real values from the robot to display the state of the game.

Live-mode: In Live-mode the EuroSim used the actual position and sensor values retrieved from the robot in action. This was particularly helpful during the test face of the robot. A socket communication was used to communicate between the robot, this is discussed further in section 3.3

Simulation-mode: The simulation mode was used to develop and test the A.I. Since the robot was still under construction, adjustable parameters and variables where used to represent speed, rotation, speed of grippers and other physical attributes. These parameters were updated when we were able to obtain real value data from testing the actual robot. Below some of the calculations used for simulating the robot will be discussed.

4.4.1 Driving

Since the robot would spend most of its time driving around the map, the simulation of movement was very important. It was determined early in the robot building process that a movement involving a change of direction would be separated into two steps. So any movement that required a rotation would first rotate, and then move in the targeted direction. This strategy was reproduced in the simulator.

$$RotationTime(phi, phi2) = r/Vr + a * Pr) \tag{4.2}$$

Here phi is the initial rotation of the robot, and phi2 is the goal rotation. Vr is the rotational velocity of the robot and Pr is translational velocity of the robot.

$$DrivingTime(p1, p2) = D/Vt + a * pa \tag{4.3}$$

$$Time(P1, P2) = RotationTime(P1phi, P2phi) + DrivingTime(P1, P2) \tag{4.4}$$

4.4.2 Obstacles

In this section we will only consider moveable game objects as obstacles. All other static features of the game environment that could be seen as obstacles are handled by the pathfinding algorithm [Section Pathfinding obstacles]. Collisions with obstacles are calculated with vector distances and intersections.

4.5 Path Planning

The path planning algorithm implemented was based on PRM described in section 2.3.3. The PRM operates on a NavMesh, a mapping of the collision-free areas of the map.

4.5.1 Constructing the NavMesh

The NavMesh is initialized and constructed from a tile representation of the static map. The static map was modelled after the board game described in [6]. For each tile a space is added to the NavMesh if the tile is clear of obstacles. The fewer spaces a navmesh consists of, the more efficient the pathfinding is. The list of spaces are therefore reduced by merging them into larger rectangles. Finding a path between two points in a navMesh can be reduced to finding a path between two spaces.

4.5.2 Obstacle Avoidance

Since the robot's size is vastly bigger than a tile, we have to make an additional effort to ensure that the robot can travel without colliding. If we consider the center of the robot as the coordinate position, there will an area around the point with the radius of the robot as distance that also can collide with obstacles. This problem can be handled directly during each pathfinding process. However, a far more efficient and simpler approach is to build it into the NavMesh. Each obstacle in the static map is extended with r + e, where r is the radius of the robot and e is an error margin optimized during testing. Including this in the static map means that we know can use navMesh to find a path between two points that guarantee a clear path for the entire robot.

4.5.3 Optimizing

As discussed earlier, finding a path between two points in different spaces include traversing the links of spaces until a valid path is found. However, there are cases where this path can be optimized. Using a line of sight algorithm, we can check for unnecessary points in a path. Consider the path a -> b -> c where each point is in a different space. If there exists a line of sight between a and c where every point of the line is in a valid space, b can be removed from the path. The difference in travel distance gained might not be the greatest, however a smoother path with fewer points will increase the travel speed for the robot discussed in this thesis immensely, since each separate rotation is a heavy time consumer.

4.6 The Opposing Robot

In the Eurobot competition there is always two robots competing against each other increasing the complexity of the original vehicle routing problem drastically. Avoiding collisions with the opponent will be one of the greater challenges for the AIs strategy.

4.6.1 Long-term avoidance

The long term avoidance will mainly be handled by the GOAP strategy and the decision making strategy. All objects are weighted with a distance from opponent factor, and the aim is to get a value function that will deevaluate objects close to the opponent.

4.6.2 Short-term avoidance

Avoiding collisions is a high priority, and a seperate emergency stop system is on top of the strategy system. The emergency system uses distance sensors in front and at the back of the robot. If the sensors report a threat higher than v, the robot will go into a emergency stop state. From an emergency state, the robot must calculate a new plan, including the opponent robot in the NavMesh.

Chapter 5

Experiments and Results

5.1 Overview

The main focus of this thesis is to investigate how reinforcement learning as a decision making strategy can be applied to a mobile robot. The goal for the experiments conducted in this section is to find a decision making policy that the GOAP-planner described in section 4.2 can utilize. The policy will be used as a value function, ranking different actions against each other in order to find the optimal plan. The policy will be trained with an implementation of SARSA described in section 2.2.4 with value function approximation.

The decision making policy will always choose the action highest scored by the trained Q-value function as described in 4.3. The experiments will involve simulating different scenarios and evaluate how different learning environments affect the performance of the decision making policies.

In the first experiments we will investigate how the decision making policy can perform on a static playing field. There are no opponents, and no time limits. In the second part we will introduce situations that can occur during a match with a changing environment.

5.2 Experimental Setup

For the Eurobot task there are multiple game objects that the robot needs to gather. For simplicity we will only consider the problem of collecting boxes and delivering them to the stand areas on the playing field. This can easily be extended to include other game objects on the final implementation since the experiments will mainly involve planning driving routes. A simplified version of the simulator described in section 4.4 was used to run the experiments.

5.2.1 Experiments on the simulator

The EuroSim simulator from section 4.4 was designed to show the performance of the robot live and simulate in real time. During the experimental phase the simulator was simplified in order to run fast simulations. Instead of a driving system based on time, a grid based movement system was implemented, where the robot would move up to one grid per update.

5.2.2 Robot configurations

For these experiments the capacity of the robot was set to 5 boxes. It was also limited to only make one delivery per stand. This limitation does not exists in the real competition, however during the planning phases of this project the plan was to build a tower of 5 boxes, and therefore the algorithm should be optimized for this setup.

5.2.3 Eurobot playing fields:

In order to investigate the performance of the algorithm, multiple permutations of the original Eurobot playing field setup was used for the experiments. Only using one playing field could lead to misleading results. The different size used for the experiments can be found in table 5.1. Multiple variants of the different sizes were used. The boxes were randomly placed for each configuration.

Name	Number of Boxes	Number of Stands	Size
Small	6	1	10x10
Medium	14	2	30x20
Large	50	7	60x40

Table 5.1: Playing field configurations

5.2.4 Simulations

In all simulations a policy was trained over 30 000 episodes. An episode is a complete game from start to finish. The number of runs vary between simulations.

5.2.5 Learning parameters

As described in 4.3.3 the learning parameters for the SARSA algorithm are case specific and had to be found with testing. Two different variants were used as seen in table 5.1

Name	Learning rate	Discount rate	Rho
Normal	0.1	0.6	0.05
Multiple features	0.3	0.6	0.05

Table 5.2: Learning Parameters

5.2.6 Rating policies

Two different rating values are used for these experiements. The first one, performance, is based on the rewards that the SARSA implementation uses to update weights.

$$Pf = 30 * Sb1 + 30 * Sb2 ... 30 * Sbn + 10 * B - 0.5s \tag{5.1}$$

Where Sb is the count of boxes at a given stand, B is the total count of boxes gathered and M is the total count of steps.

The second rating is called Game score and is based on the scoring system of a Eurobot match. There are no points given for boxes not delivered, and there is no step penalty.

$$Gs = 30 * Sb1 + 30 * Sb2 .. 30 * Sbn \tag{5.2}$$

5.2.7 Analyzing results and policies

Policies
In order to analyze the different policies learned by the robot, a pcolor plot as seen in figure 5.1 will be used. Each column is a trained policy learned over multiple episodes of training. Yellow boxes indicates a weight with high value, a blue bloxes indicate a negative value. A learning configuration that produces stable policies are indicated by a weight being consistent for all runs.

5.3 Static planning

In this section we will try to find a good static policy for the eurobot task with value function approximation. We will use a simplified version of the playing field, where only target is to gather and deliver boxes to the stands. All solutions will be measured in the performance evaluation from section 5.2.6. The goal is to create a policy that will select effective paths based on the information given from environment. We will also investigate if a general policy can be found for multiple playing fields. It is not likely that we will find a policy better than proven dynamic and genetic algorithms for similar problems, however a good static algorithm will be the ground layer for our dynamic planning algorithm.

5.3.1 Feature extraction

The first experiments conducted involved comparing different variants of feature extractors against each other. A RL algorithm can only learn from

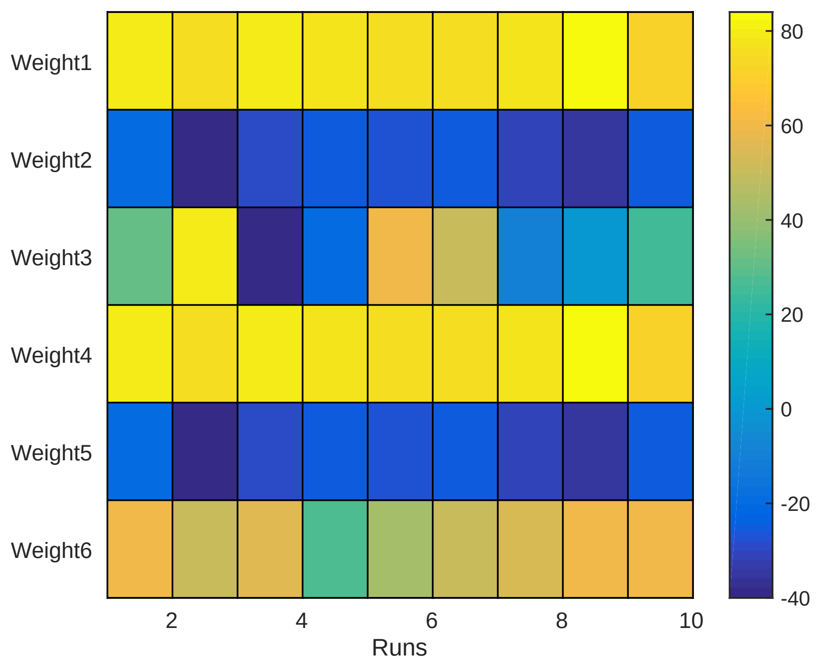

Figure 5.1: Example plot of trained policies. Each column represents policy.

the information represented in state. In value function approximation, the features represents the state of the environment. In order to find the optimal feature extraction, multiple experiments were done with different combinations. The feature extractions were tested on multiple playing fields of different size as described in section 5.2.3. Below is a short description of the different variations. It was expected that the ActionStates varation would have the best performance, because it contained most information about the environment.

PosBoard: Uses a combination of position and board state. Features 1..4 for position and 5..8 for board state.

ActionStates: Expanding PosBoard with details about the next action. Extracts distances to nearby objectives of the current action.

NoPosition: Removes position features 1-4 and only focus on board state and action states.

Parameters The parameters for the Q-value update rule explained in section 4.3.3 can change the outcome of a value function approximation drastically. Based on experiences from other projects the parameters were initially set with a learning rate of 0.7, discount rate of 0.3 and rho of 0.2. These parameters achieved good results on smaller maps. When the complexity of the problem increased with larger maps, the feature vector

Feature extractor	Small 1	Small 2	Medium 1	Medium 2	Large 1	Large 2
PosBoard avg	**111**	117	**172**	150	458	405
ActionStates avg	100	117	143	**161**	**523**	**444**
NoPosition avg	92	117	171	155	487	366

Table 5.3: Performance average of feature extractors

Feature extractor	Small 1	Small 2	Medium 1	Medium 2	Large 1	Large 2
PosBoard max	111.0	117	176	159	510	522
ActionState max	111	117	176	175	610	482
NoPosition max	111	117	176	175	578	400

Table 5.4: Best performance of feature extractors

weights would tend to grow indefinitely. By decreasing the learning rate to 0,1 the weights stabilized. By decreasing the learning rate, earlier experiences have less value. This resulted in policies that found a good solution halfway through the learning period and at the end changed course and ended up with a worse policy. To make up for the low learning rate, the rho was reduced to 0,05.

Results:
As table 5.3 and 5.4 shows, the assumption that ActionState would have the best performance hold up. The variation achieved the best overall performance on average and on top performance. On the smaller playing fields, all extractors found a decent policy, and was able to find the optimal solution on at least one of it runs. When the complexity of the problem increased, the ActionValue was able to use the extra information it contained in order to find a better policy. As seen on Figure 5.2, the ActionState and NoPosition had similar results, which was the case for all small and medium problems. However on Figure 5.3 ActionState is clearly able to find a better policy than the other variations when the complexity of the problem increases.

Analysis
As described in section 5.2.7, the columns in figure 5.4 are learned policies and rows are the values of weights. In figures 5.4 and 5.5 we can see that the policies are consistent for all runs. We can also see that the two different variations have found the same policy. Both have given a high value to BoxesCarrying and AvailableBoxes, and a negative value to TimeActionComplete and DistanceToBox1. From this we can conduct that the policy will prioritize boxes and stands that are close to other boxes, and choose boxes it can reach quickly. It is interesting to note that AvailableBoxes has such a high value. Since a low amount of available boxes indicates that the robot has been able to collect more boxes and thus score more points, you would assume that this weight would be negative. A possible reasoning for this could be the high discount factor. In the beginning of a round, the count of available boxes is high, and the potential future value for the robot

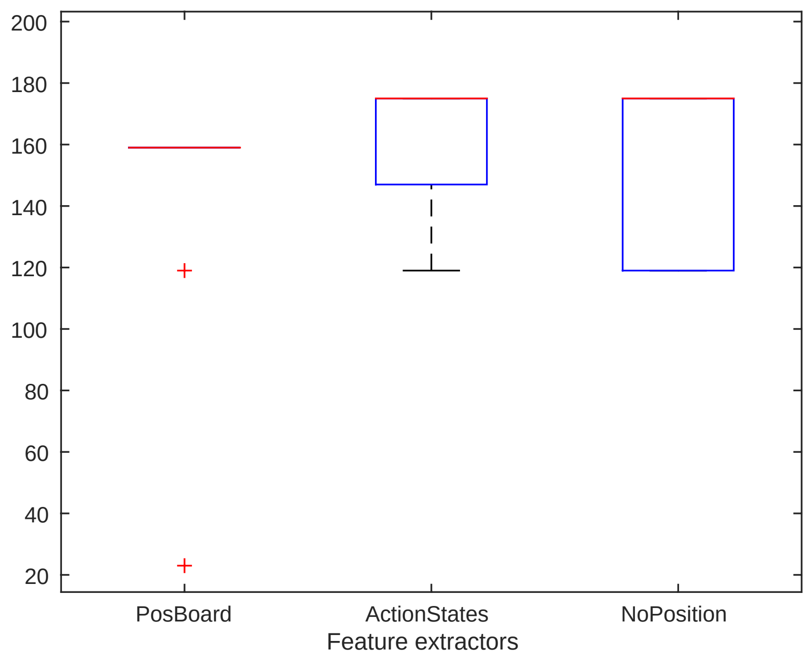

Figure 5.2: Box plot of performance on Medium 2

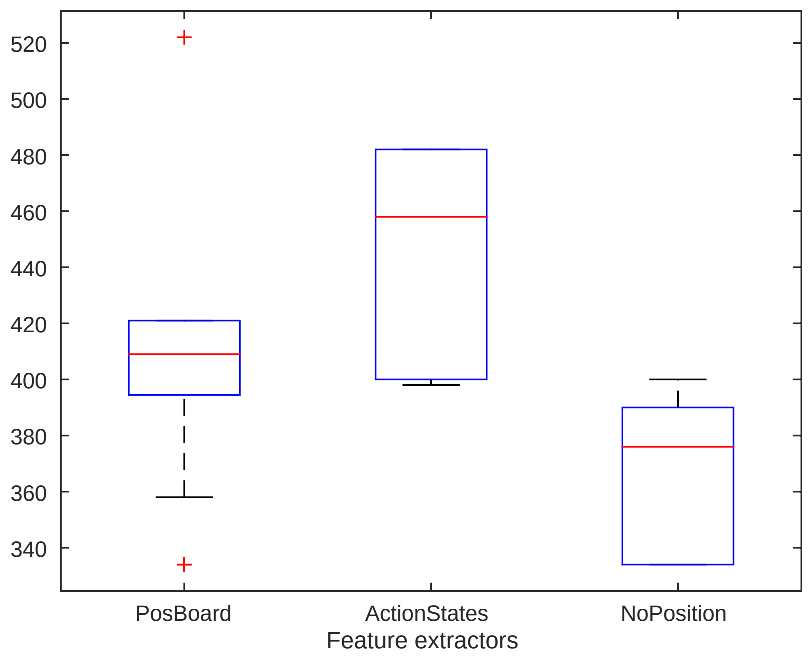

Figure 5.3: Box plot of performance on Large 2

to score is also high. Therefore the policy could take this as an indication that a high value for AvailableBoxes is the sign of a good policy. Distan-

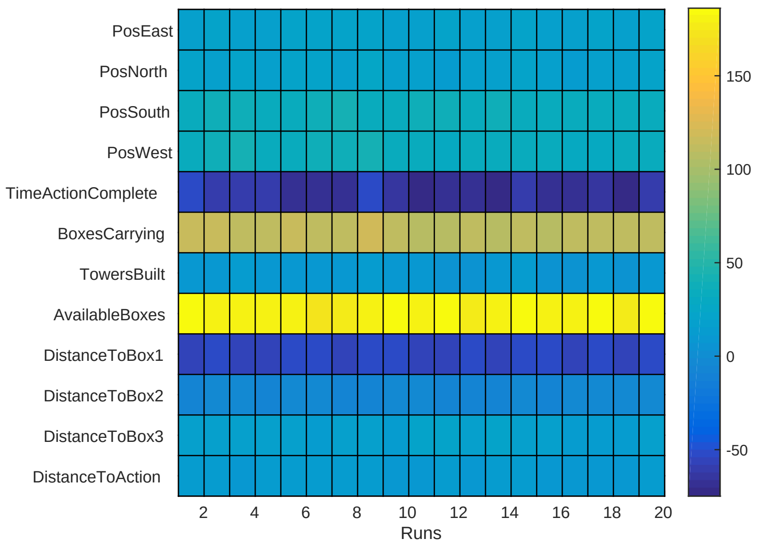

Figure 5.4: Weight distribution for policies with a ActionStates feature extractor over multiple runs.

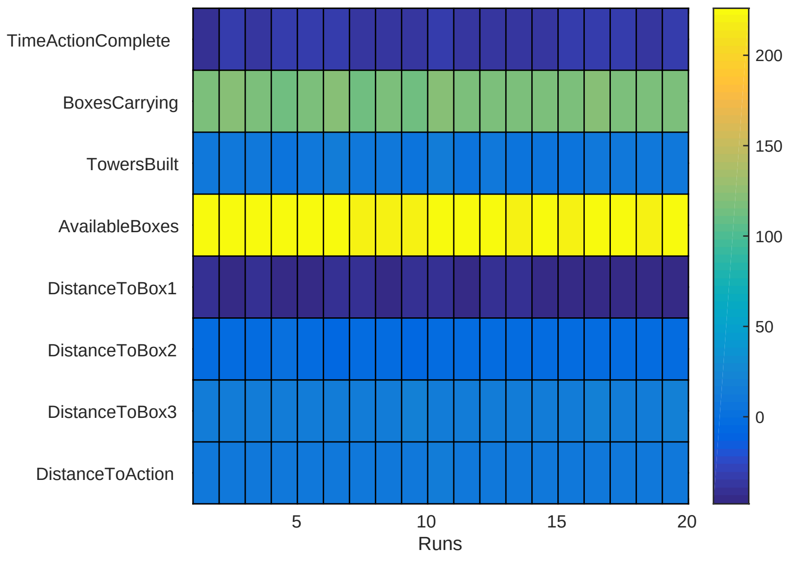

Figure 5.5: Weight distribution policies with a NoPosition feature extractor over multiple runs.

ceToBox2 and DistanceToBox3 are mainly ignored, which indicates that the policy is not able to utilize this information. It is interesting to note how big of a difference it is between the ActionStates and NoPosition extractors. The policy is able to utilize the positioning of the robot on the different playing field in order to find better routes. In figure 5.4 we can see that posSouth and posWest have a higher value than posNorth and posEast, which indicates that there is a higher value of being in the North-East corner on that map.

5.3.2 Training on multiple maps

The second investigatory experiment involved testing policies trained on a single map vs policies trained on multiple maps. This was done in order to examine if a policy trained on multiple maps was able to find a general policy, and how this policy would hold up against the other variations. Based on previous results the ActionPos feature extractor will be used for all variants. The smallest maps showed no interesting results in the previous experiment, and have been replaced with two new mediums. It is not necessarily the goal of this experiment to find a policy that works well on all map configurations. However it is interesting to see if the assumption that a policy trained on a single map is able to use the positioning features to its advantage. A general overview of the different variants is given below.

Trained on single perform multiple:
The agent is trained on a single map and will perform on all. The goal of this experiment is to see if the agent is able to find a map specific policy that is better than a generalized policy.

Trained on multiple, perform multiple:
This agent is trained on all map configurations. During training, a random map is drawn from the pool of map configurations.

Trained on	Medium 1	Medium 2	Medium 3	Medium 4	Large 1	Large 2
Medium 1	119.5	165.0	165.0	124.5	565.8	517.5
Medium 2	176.0	155.4	201.0	169.0	610.0	554.0
Medium 3	172.0	119.0	158.4	98.0	610.0	523.0
Medium 4	176.0	152.6	197.7	162.2	610.0	554.0
Large 1	9.0	21.0	17.0	17.0	516.1	404.1
Large 2	9.0	21.0	17.0	28.2	574.0	450
Multiple 60k	135.5	139.8	170.4	140.5	365.4	343.3
Multiple 20k	112.0	152.2	192.2	165.0	610.0	554.0

Table 5.5: Trained on single map vs multiple maps average

Results From table 5.5 and 5.6 we can see that in general the variations trained on multiple maps have a fairly similar score to the variations trained on single medium maps. This indicates that a general policy might works just as good as any map specific information learned by variants training on a single map. However it is interesting to note how poorly the variations trained on large maps perform on medium maps. As the multiple variation is also trained on the large map, this indicates that it is better at finding a general policy for all problems than the map specific variations.

Trained on	Medium 1	Medium 2	Medium 3	Medium 4	Large 1	Large 2
Medium 1	176	175	201	169	610	554
Medium 2	176	175	201	169	610	554
Medium 3	176	119	179	107	610	554
Medium 4	176	167	201	169	610	554
Large 1	9	21	17	17	610	482.4
Large 2	63	47	17	45	610	482
multiple 60k	176	168	196	186	610	554
multiple 20k	145	165	201	165	610	554

Table 5.6: Trained on on single map vs multiple map best performance

Analysis: In figure 5.6 we can see the weights for the variation trained on multiple maps. The policy is mainly based on finding boxes that are close to other boxes and in short distance from the robot. This can be seen on the negative weights for DistanceToBox1 and TimeToCompletion. AvailableBoxes is the most positive weight. A possible reason for this might be that in the early phases of a game, there are a lot of potential points to score and the available boxes feature is high. Therefore the algorithm believes that this is a sign of a good policy. However since the weight for DeliverBox actions, this can lead to early deliveries which would be a bad policy. This is a common problem when operating with a single weight vector for a value function approximator. In the next section we will experiment with separating weights for actions to resolve this problem. Overall we can see that training on multiple maps does not lead to a drop in performance for the static solver.

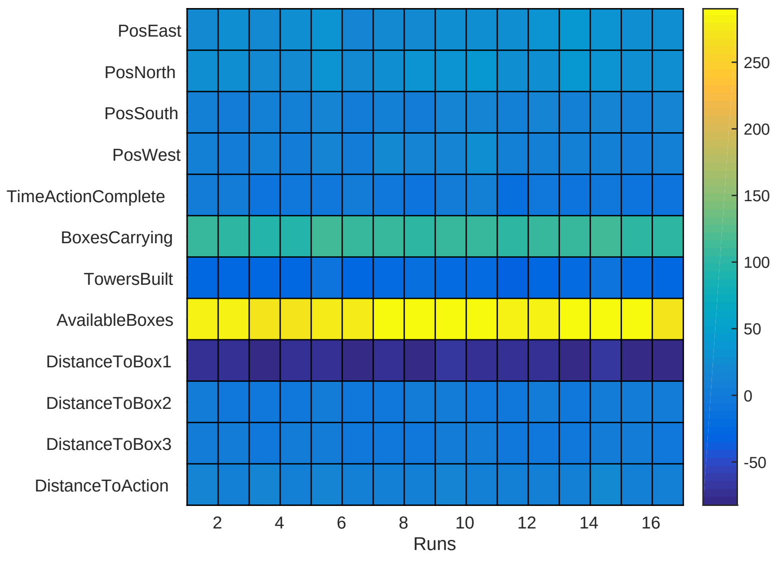

Figure 5.6: Weights after training on multiple maps

5.3.3 Train with multiple feature weights

In previous experiments we were only using a single feature weight vector for all actions. In this investigatory experiment we will test the performance of policies with multiple weight vectors. A common approach is to create a sepearate feature weight for each action. A feature that is very important for one action, might not be relevant for another.

The multiple feature weights variants will be tested by both training on a single map and multiple maps.

Trained on	Medium 1	Medium 2	Medium 3	Medium 4	Large 1	Large 2
Medium 1	147.0	165.9	151.8	154.0	504.5	482.5
Medium 2	147.9	163.9	145.3	141.2	441.0	415.6
Medium 3	151.5	165.7	144.7	143.6	486.4	429.9
Multiple	147.5	161.9	153.5	144.4	414.2	396.8

Table 5.7: Trained with multiple feature weights performance average

Trained on	Medium 1	Medium 2	Medium 3	Medium 4	Large 1	Large 2
Medium 1	166.0	187.0	201.0	169.0	610.0	554.0
Medium 2	166.0	187.0	155.0	161.0	480.0	451.0
Medium 3	166.0	187.0	155.0	161.0	514.0	465.0
Multiple	167.0	187.0	196.0	186.0	530.0	498.0

Table 5.8: Trained with multiple feature weights highest performance

Parameters: In order to change from a single weight vector to a weight per action implementation, changes had to be made to the parameters. On early runs with the parameters from previous experiments the weight values would grow indefinetly. In particular, the weights for DeliverAction would with the current reward system only get bonuses, and grow indefinitely. In order to stabilize the weight vectors, the algorithm was tested with new parameters. After testing with different values it was found that increasing the learning rate to 0,3 yielded the most stable weights.

Results
Comparing table 5.7 and 5.5 we can see that separating weights for actions did not lead to a improvement of performance from the previous experiment.

Anaysis: However, it is interesting to note the stability of policies over multiple maps. In the last experiment we could see that policies trained on a single map had huge variances. For example Medium 1 in table 5.5, the policy did not perform well on the map it was actually trained on, but on Medium 2 and Medium 3 the performance was good. In this experiment we can see from 5.7 that the performance for all policies did not vary significantly, and did not have a drop for certain maps.

The learned policies can be seen in figure 5.9 and figure 5.10. We can clearly see that the policy has weighted the actions differently. The pickBox weights seen in figure 5.9 have given a high value to boxesCarrying and availableBoxes, and a negative value to distanceBox1. The values are very similar to previous policies with a single weight vector. On the other hand, DeliversBoxAction has weighted towersBuilt as high and AvailableBoxes low. We can here see that seperating the weights has allowed the policy to find a customized weights vector for PickBoxAction that fits better. With AvailableBoxes weighted as a negative, the policy will prioritize gathering more boxes before choosing to deliver the boxes. However this has not affected the overall performance significantly.

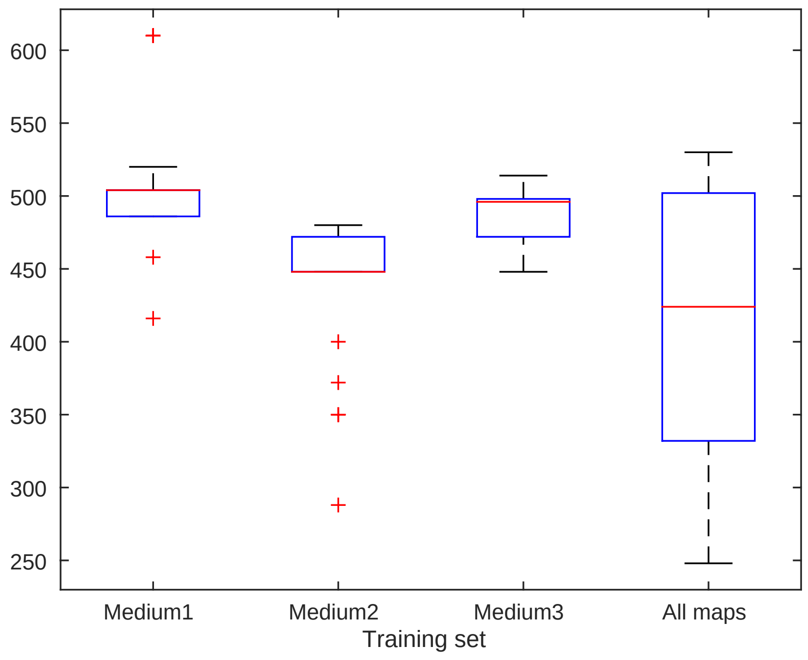

Figure 5.7: Weights for PickBoxAction when training with multiple features

5.3.4 Performance

In the previous experiments we have tested four different training variations. In this section I will give an overview and discuss the results found. Table 5.9 is an overview over all the different setups. The different variations are described below. The variations marked as single are both trained on the first map, Medium 1.

SingleMapSingleFeature (SMSF):
Trained on one map with one feature table.

MultipleMapSingleFeature (MMSF):

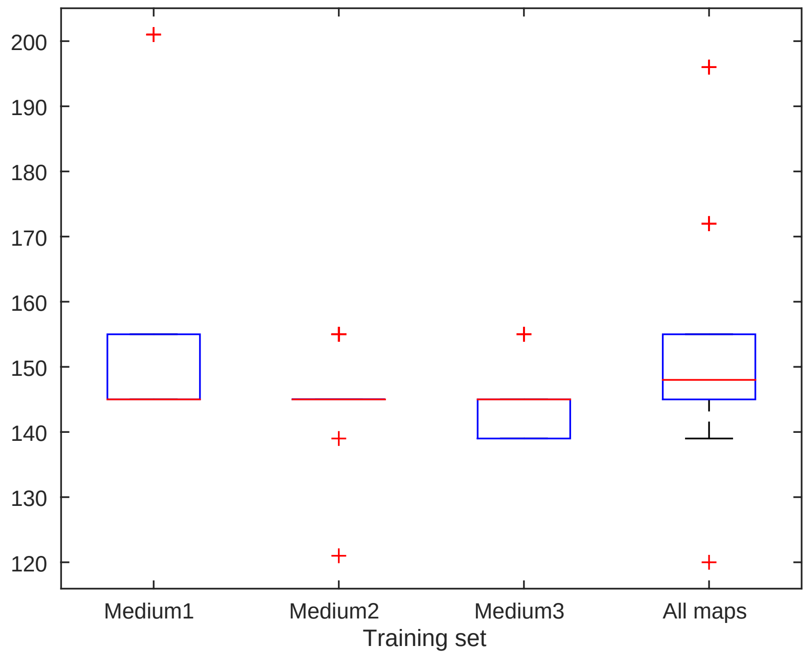

Figure 5.8: Weights for DeliverBoxAction when training with multiple features

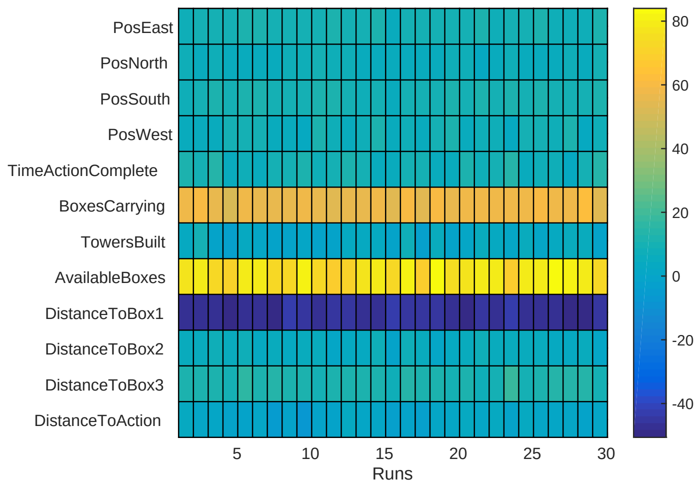

Figure 5.9: Weights for PickBoxAction when training with multiple features

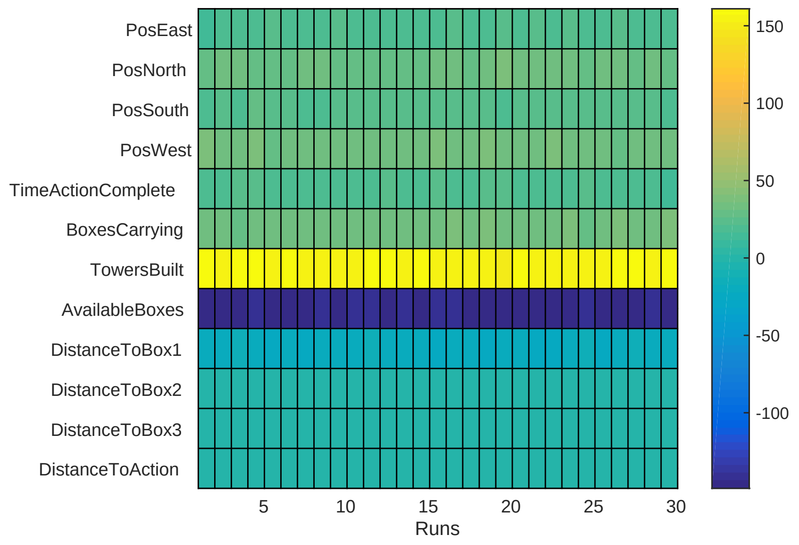

Figure 5.10: Weights for DeliverBoxAction when training with multiple features

Trained on multiple map with one feature table.

SingleMapMultipleFeatures (SMMF):
Trained on one map with multiple features, one for each action.

MultipleMapMultipleFeature (MMMF):
Trained on multiple map with multiple features.

Variation	Medium 1	Medium 2	Medium 3	Medium 4	Large 1	Large 2	Total
SMSF M1	119.5	165.0	165.0	124.5	565.8	517.6	1657.4
SMSF M2	**176.0**	155.4	**201.0**	**169.0**	**610.0**	554.0	**1865.4**
MMSF	112.5	152.2	192.2	165	**610.0**	**554.0**	1785.9
SMMF	147	**165.9**	151.8	154	504	504.5	1627.2
MMMF	147.5	161.9	153.5	144.4	414.2	396.9	1418.4

Table 5.9: Overview of performance for the different variations used for the static solver.

Discussion Based on the results from the experiments conducted, none of the variants has had a significantly better performance than the other. As we can see in table 5.9 the average performance of SMSF trained on the Medium2 has overall the best performance for a general policy that works for multiple map configurations. However, the same SMSF variation trained on a different map Medium1, does not achieve the same level of performance. We have seen that all variants have been able to find stable and reasonable policies. The variant with multiple features developed the most advanced policy. However, this did not reflect in a significant increas of performance.

5.4 Experiments on a changing environment

In this section we will build up on the different learning variants from the previous experiments and introduce challenges that comes with a changing environment. We will introduce a time limit and an opponent to see how the different variations adapt. The configurations from 5.3.4 will be the foundation for the next experiments, and a few new variants will also be introduced. In the last experiment, the policies will be test on specific cases related to the Eurobot task.

5.4.1 Delivery on time

The robot will only get points for the boxes that are delivered to a stand. During a game there are endless possible situations that can occur where the robot ends up being stuck or blocked by the opponent robot, making the optimal plan impossible to execute in time. Therefore it is important that it learns to prioritizing delivery before the time runs out. However we do not want a policy that misses points by delivering early just to be on the safe side. In this experiment we will test on maps with time limits. Two different medium maps will be used with three different time limits; 70, 120 and 160.

Bonus
Variations marked as bonus are given a higher performance reward for building maximum stands. This was introduced in order to see if it would help the algorithm perform at maximum when it had time. This is not reflected in the table, as the game score is not effected by this.

Table 5.10: Early delivery average

Setup	M1 70s	M1 120s	M1 160s	M2 70s	m2 120s	m2 160s
SM-SF	112.4	85.8	71.4	180	97.8	64.2
SM-MF	126	190.2	198	142.8	205.8	214.2
SM-SF-Bonus	129.6	97.8	111	168	118.8	106.2
SM-MF-Bonus	118.8	183	209.4	143.4	211.2	207.6
MM-SF	117.6	169.8	236.4	178.8	184.8	203.4
MM-MF	127.8	155.4	181.8	127.2	220.8	241.8

Result The average result of this experiment can be seen in table 5.10 and best policy achieved in 5.11. As mentioned in the description, results for this experiment are measured in goal score 5.2.6 and not in performance as previous experiments. From table 5.10 we can see that the SMSF variants had a significant drop in performance.

Analysis A problem with the variants training on a single map, is that the

Table 5.11: Early delivery best run

Setup	M1 70s	M1 120s	M1 160s	M2 70s	m2 120s	m2 160s
SM-SF	180	210	240	180	180	180
SM-MF	150	270	300	180	300	300
SM-SF-Bonus	210	270	300	210	180	300
SM-MF-Bonus	150	300	300	180	300	300
MM-SF	150	300	300	180	300	300
MM-MF	150	270	300	210	300	300

playing field has to reflect all the challenges in order for the policy to learn. A possible reason for the drop in performance could be that the training map for SM had a time limit that was too low, and that the policies were too focused on delivering early. The performance drop is significantly lower compared to other policies on the 120s and 160s maps. For future experiments the time limit of the training map should be set higher.

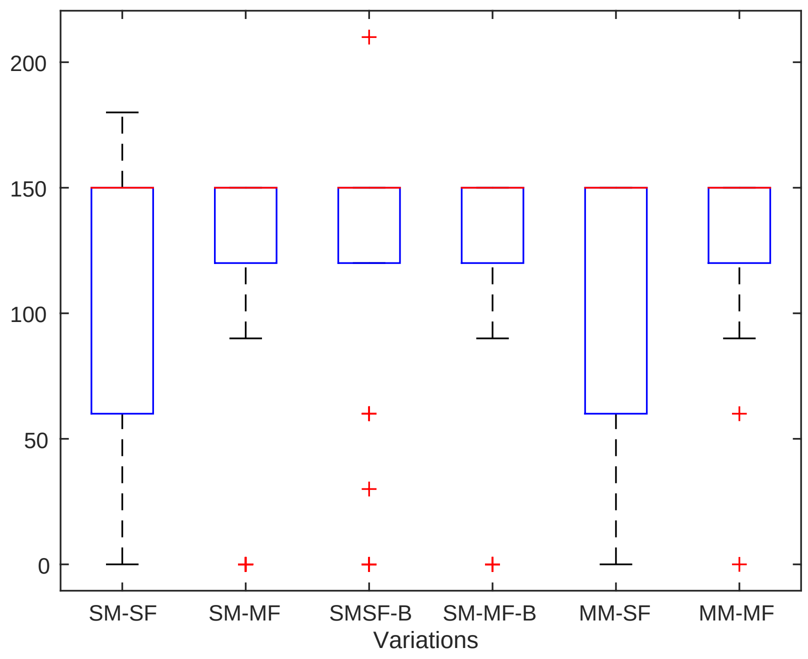

Figure 5.11: Results with maximum time 70

5.4.2 Avoiding the opponent

Avoiding collisions is one of the main objectives for the A.I, and therefore this should be considered during the planning face. By choosing boxes far from the opponent, we are less likely to avoid collisions. In this investigatory experiment we will try out different strategies for avoiding boxes that can lead to collisions. The playing fields are in this experiment set up with an opponent next to one of the boxes. If the robot picks this box, it will

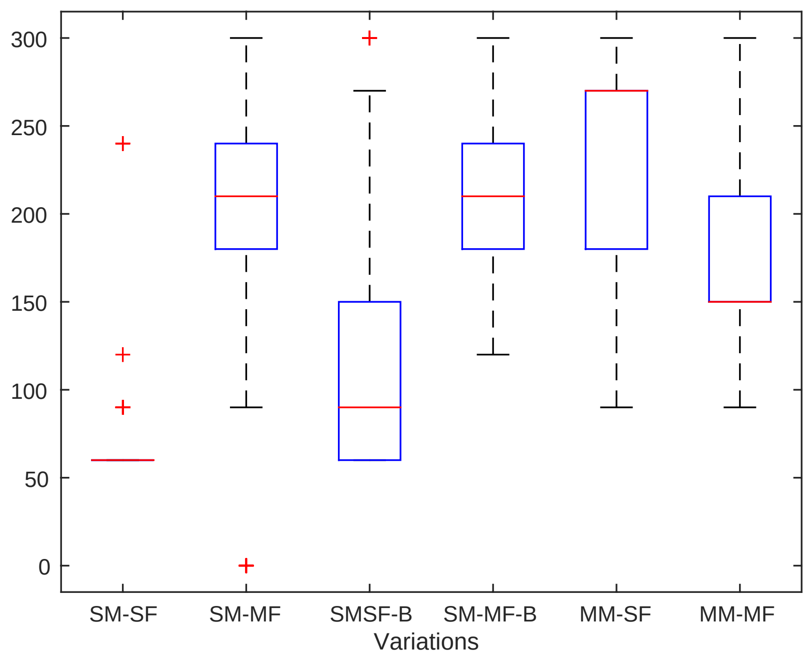

Figure 5.12: Results with maximum time 160

crash and the game is over. In order for the robot to reason about the position of the opponent, we must add information about this to the state and feature representation. Two different implementations are tested in this experiment. A short description follows below.

StaticThreat If the opponent is within x range of a box, this feature will be set to 1, indicating that a threat is near.

DynamicThreat This feature consists of the distance to the opponent robot. In order to stabilize it a maximum distance of l is set.

NoDetect Does not have any information about the opponent.

The different feature representations are tested on three different maps with two different robot positions. Four different training techniques are used, similar to previous experiments.

Results From table 5.12 we can see that the NoDetect variation without information about the opponents position is not able to adapt its policy and avoid crashes. The variants with a feature representing information about the opponent were all able to adapt and find a policy where crashing was avoided. However, seen in 5.13 and 5.14, all variants had some policies that lead to a crash.

Analysis In figure 5.16 and 5.15 we can see an overview of the different

Variation	map1_1	map1_2	map2_1	map2_2	map3_1	map3_2
SMSF-NoDetect	111	48	133.2	78	181.8	1.2
SMSF	208.2	**242.4**	144.6	127.8	179.4	210.6
SMSF-static	214.2	220.8	**214.8**	216.6	231.6	138
MMSF	197.4	197.4	186.6	154.2	**238.8**	**220.2**
MMSF-static	172.2	188.4	205.8	**223.2**	213.6	135.6
SMMF-static	198.6	196.2	195.6	198	198.6	130.8
MMMF	204.6	178.8	205.2	205.8	208.2	218.4
MMMF-static	199.2	186.6	208.2	206.4	163.2	115

Table 5.12: Avoiding opponent average

Variation	map1_1	map1_2	map2_1	map2_2	map3_1	map3_2
NoDetect	210	150	240	300	300	30
SMSF	270	270	270	150	210	270
SMSF-static	300	300	300	300	300	270
MMSF	240	300	300	300	300	300
MMSF-static	300	300	300	300	300	270
SMMF-static	300	300	300	300	300	270
MMMF	300	300	300	300	300	300
MMMF-static	300	300	300	300	300	300

Table 5.13: Avoiding opponent best run

weights for policies trained with SFSM for both variations. We can see that the general strategy is quite similar, represented by the positive weghted features. The DynamicThreat feature is weighted high, which means that boxes far away will be valued higher than boxes close to the opponent. The StaticThreat feature is negativly valued and will punish boxes close to the opponent. All though figure 5.12 show that the different implementations have a similar performance, it does effect the strategy when the robots are competing. The dynamic implementation will imply a very defensive strategy where you will always pick boxes far away from the robot, and allowing the opponent to go for his optimal strategy. With the static threat feature you will still avoid going for objectives that may lead to a crash, but it doesnt hinder the robot from staying agressive and challenge the space around the opponent. With this in mind, the static threat feature

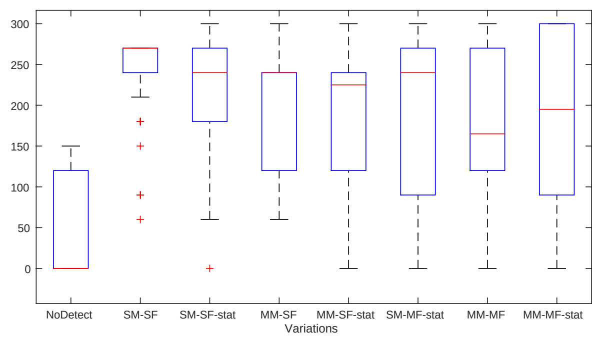

Figure 5.13: Avoiding the opponent first map second configuration

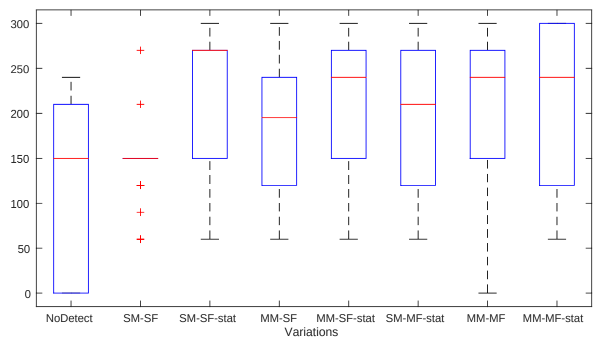

Figure 5.14: Avoiding the opponent second map first configuration

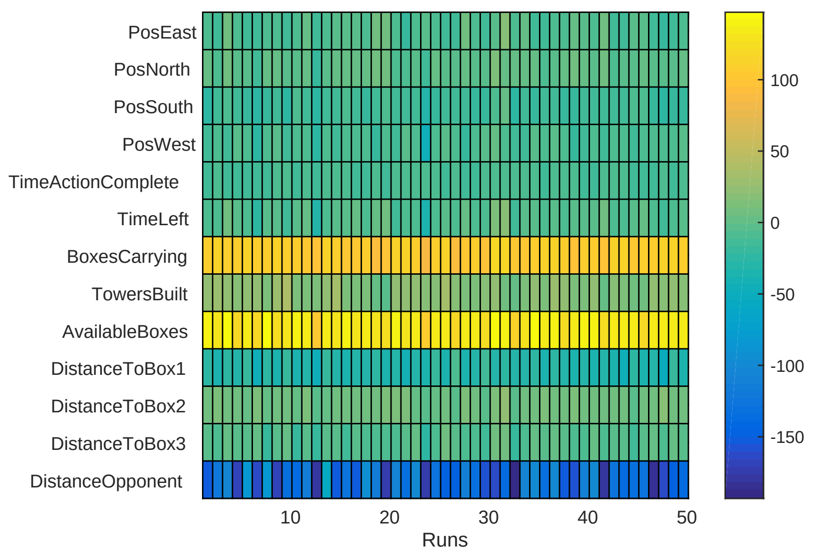

Figure 5.15: Weights when trained vs opponent with a static threat feature

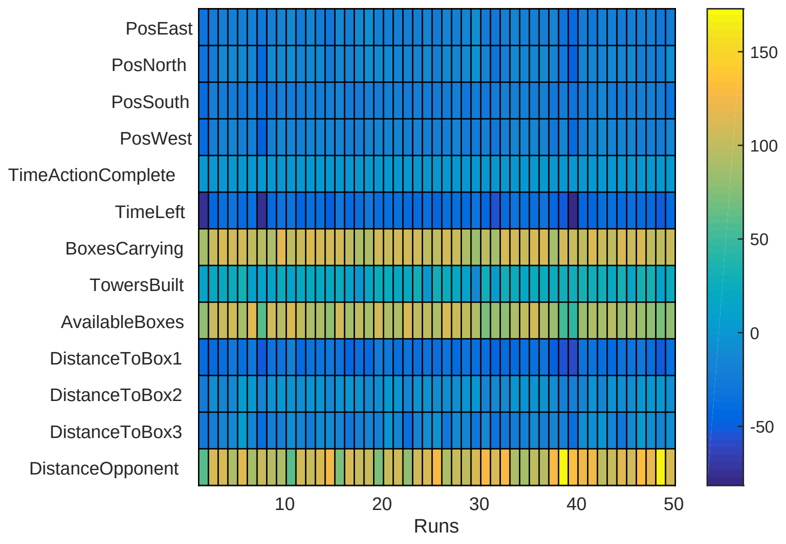

Figure 5.16: Weights when trained vs opponent with a dynamic distance to opponent feature

5.4.3 Eurobot edge cases

In previous experiments we have tested learning variations for different challenges with a changing environment. In all cases they were specifically trained to solve the challenge, by learning on playing fields focused on the task. In order to perform well on the Eurobot task we need to combine the problems and find a policy that can handle all of them. In this experiment we will train four policies and test them on four different edge cases involving a difficulties with a changing environment. We will also run the policy on a static map in order to check performance in comparison with previous results. The first two edge cases will test the policy for delivering on time. As discussed earlier the robot will only receive points for boxes in delivering areas when the time runs out. Case three and four will test the policy for avoiding boxes near the opponent that can lead to crashes. At last we will test the policy for a static solution in order to investigate how the dynamic additions has effected the static solution.

Experiment setup
From the two feature differen feature variations from 5.4.2 we will use the static threat feature. For variations training on multiple maps, a combination of playing field from **??** and 5.4.1 will be used. Variations training on a single map will train on a playing field with a medium time limit and an opponent. All cases will be explained below with an illustration and a short description.

Evaluation
A performance score will be calculated for each policy. For each case there are multiple choises the policy can take, and only one is considere bad.

Choosing a bad action will yield 0 points, all other 100. The goal score from static solution will also be added to the performance score.

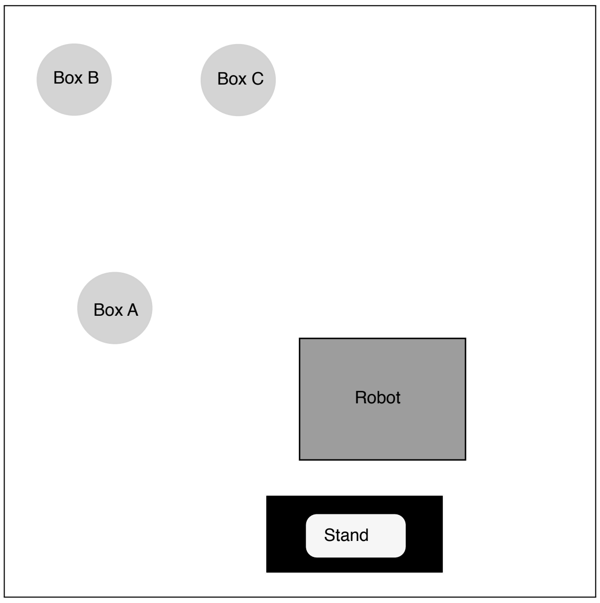

Figure 5.17: Illustration of case A: With limited time, the robot must choose between delivering or pick another box

Case A:
In this edge case the robot is standing next to a delivery area with only 5 seconds left on the clock. The robot is currently carrying 3 boxes, and now have to choose between delivery or going for another box. If the robot chooses to go for another box, it will not have enough time to deliver, and loose all points for the boxes already gathered. Illustration of the problem can be seen in figure 5.17.

Case B
This problem is a variation of the previous edge case. As seen in previous experiments, policies tend to value boxes that are grouped together. In this test one of the boxes is moved closer to the nearest box, which should make it harder for the policy to decide. An illustration can be seen in figure 5.18

Case C
Here we will test to see if the policy can avoid dangerous boxes. The robot is searching for a new box, and has three options. Box B is currently being picked up by the opponent. If the policy chooses Box B it will crash. All other boxes are valid. Illustration can be seen in 5.19

Case D
This is a variation of case C where one box is moved closer to the dangerous box. It is not within threat distance for the static feature AVOIDOPP. As in case B, moving box A closer to B should also make B more popular

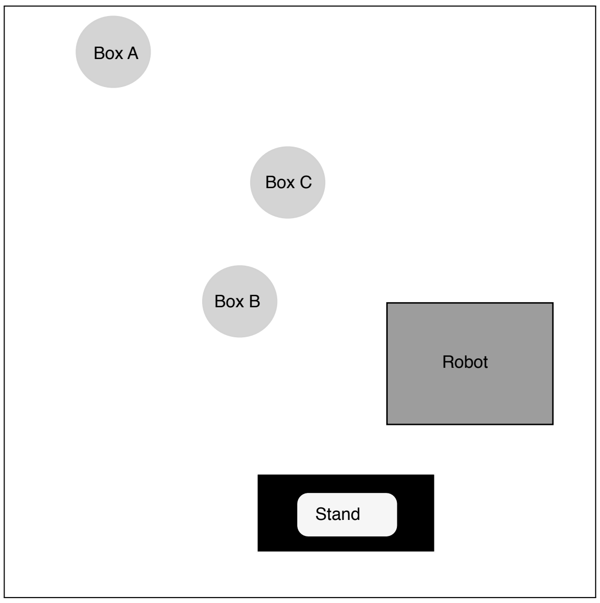

Figure 5.18: Illustration of case B: With limited time, the robot must choose between deliver or gathering another box. With two boxes close together, this decision should be harder than case A.

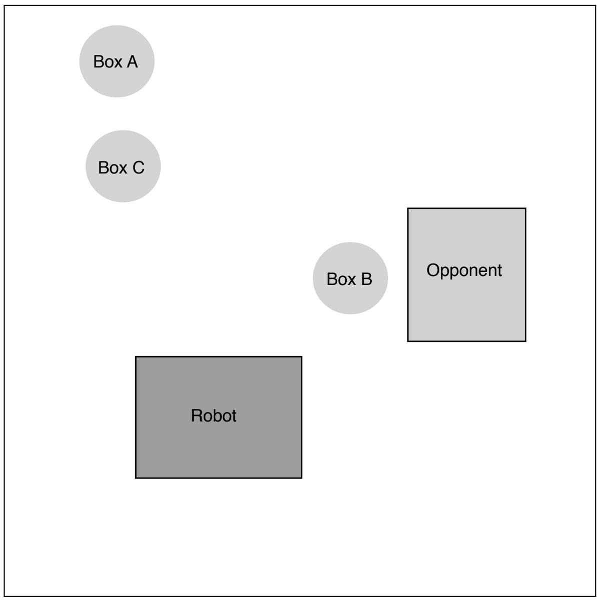

Figure 5.19: Illustration of case C: The robot must choose which box to gather next. Choosing Box B will lead to a crash with the Opponent.

with the policies.

Case E:
The last case is to find a static solution to the Eurobot task. This is to check that the policy is still able to find a good solution to the original problem while adapting to the dynamic problems. The playing field used in this ex-

Variation	Box A	Box B	Box C	Deliver
SMSF	0	0	0	100%
MMSF	27%	0	0	73%
SMMF	69%	0	0	31%
MMMF	29%	0	0	71%

Table 5.14: Results from edge case A

Variation	Box A	Box B	Box C	Deliver
SMSF	0	0	0	100%
MMSF	0	70%	0	30%
SMMF	0	70%	0	30%
MMMF	0	74%	0	26%

Table 5.15: Results for policies tested on Case B

Variation	Box A	Box B	Box C
SMSF	13%	8%	79%
MMSF	53%	2%	45%
SMMF	33%	15%	52%
MMMF	48%	7%	45%

Table 5.16: Results for policies tested on case C

Variation	Box A	Box B	Box C
SMSF	85	7	8
MMSF	97	3	0
SMMF	12	18	70
MMMF	91	7	2

Table 5.17: Results for policies tested on case D

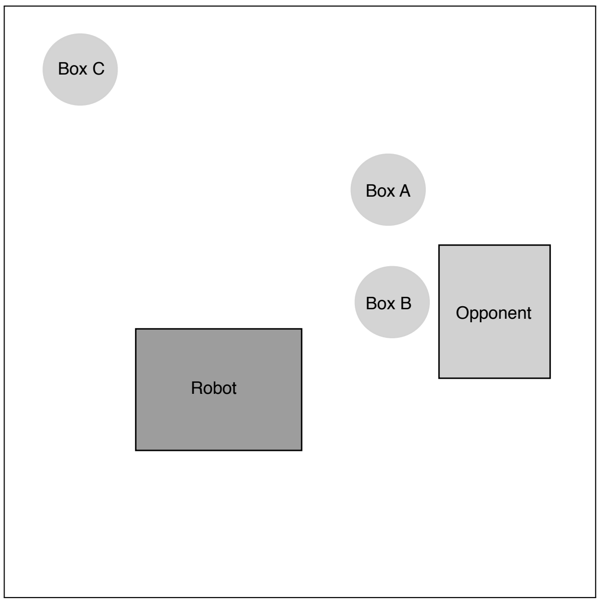

Figure 5.20: Illustration of case D: The robot must choose which box to gather next. Choosing Box B will lead to a crash with the Opponent.

periment is a replica of this years Eurobot task. There is no opponent, and no time limit on this experiment.

Results From table 5.22 and graph 5.22 we can clearly see that the SMSF variaton achieved the best performance, withith an average of 630 out of a maximum 700. SFSM was able to choose the correct solution 100% of the times on the two time limit tests in case A and B. It was also consistently avoiding the opponent by choosing the right box over 90% of the times on case C and D. It also had the highest average performance for the static tast as seen in figure 5.21. MMSF and MMMF were both able to obtain a good average on case A, with choosing to deliver 70% of the times. However, when the two boxes were grouped up for Case B, the performance dropped to 30%.

Analysis
The policies learned over multiple runs for the winner SMSF can be seen in figure 5.23. We can see that with the current learning setup, the algorithm was able to find a very reasonable policy. The policy has weighted TimeActionCoplete low and TimeLeft high, which indicates that it will punish actions that are time consuming. BoxesCarrying is weighted high amd AvailableBoxes is weighted low which indicates that the policy will try to gather as many boxes as possible before it delivers. From the case tests, the policy showed that it would prioritize delivering when there was little time left, while still being able to complete a full run if there were time. It was also able to avoid boxes that lead to a crash, while still being agressive by going for boxes close to the opponent.

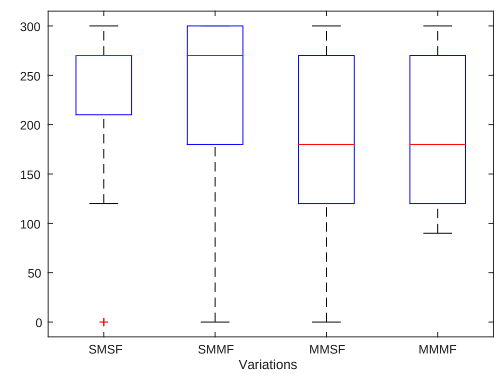

Figure 5.21: Results for static solutions on case E.

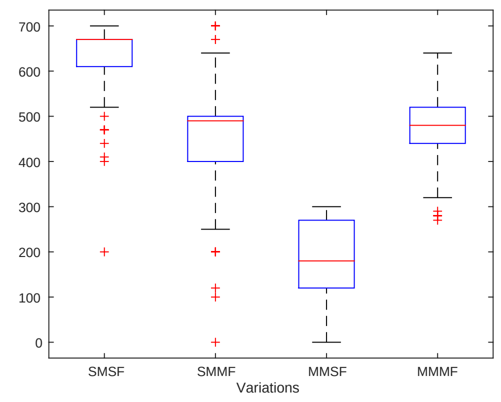

Figure 5.22: Overall performance on edge cases

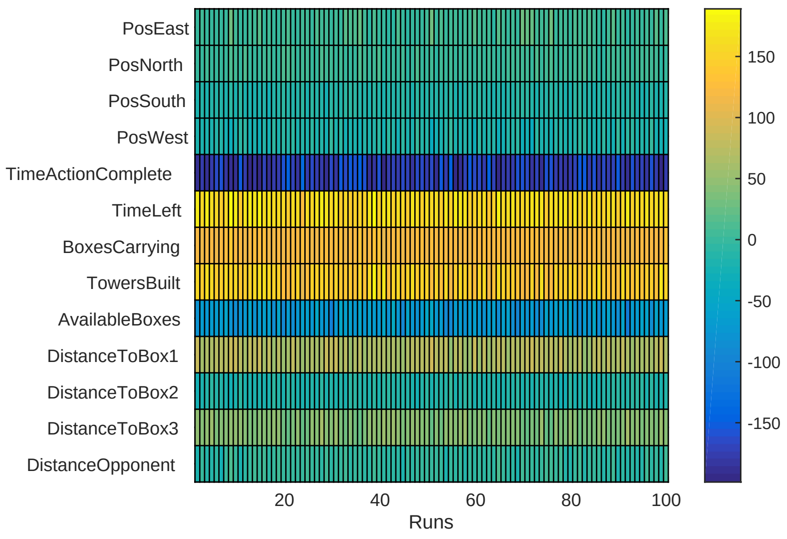

Figure 5.23: Weights for the SFSM policy.

Chapter 6

Discussion

6.1 Discussion

The main goal for this thesis was to investigate Reinforcement Learning as a decision making strategy for an mobile robot, competing in the Eurobot 2015 competition. For this purpose, a Goal Oriented Action Planner was implemented and extended with reinforcement learning in order to handle a changing environment.

The GOAP-planner used a trained decision making policy in order to rate the value of an action depending on the state of the game. The results from experiments in section 5.3 and 5.4 show that the SARSA implementation with feature value function approximation was able to create good policies for the planner on a Eurobot task. On experiments related to changing environments, the trained policy was able to adapt its strategy in order to avoid objects near the opponent and to change its strategy based on time limits. During experiments for edge cases on the eurobot task, the policy trained on a single map with a single feature weight vector achieved the best results.

A weak area of the value function approximation was the variance. Even though the final policy had a good average performance, there were some learned policies that failed completely. This indicates that we were not able to find a combination of parameters and reward structure for the value function approximation that would always lead to a good result. However, using simulation a good decision making policy could be found over multiple runs.

Unfortunetaly, because of faults with the final robot, we were not able to utilize the trained policy in the competition. Without results from the actual robot it is impossible to conclude that the trained policy would have worked.

The last subgoal stated the need for a real-time simulator. This turned out to be of singificant importance for the AI, as the robot was not finished until a few weeks before the competition. The simulator was also used for monitoring live test runs by reading the state of the real robot.

6.2 Conclusion

An AI with a decision making policy trained with reinforcement learning was presented in this thesis. The algorithm consists of a policy for action picking based on a approximated value function learned from training. The algorithm was tested in simulation with different parameters and learning configurations. The results show that the decision making policy was able to adapt its strategy to a changing environment. Unfortunetaly the AI could not be used and tested on the intended robot.

The Eurobot team was able to build a robot and compete in the Eurobot 2015 competition in Switzerland. This was the first time a team from the University of Oslo attended the competion. The robot qualified for the main tournament, and won two out of four matches played. As the main objective was to qualify, this was perceived as a good result for the team.

6.3 Future Work

6.3.1 RL-policy

Even though the decision making policies trained with reinforcement learning showed promising results in the experiments conducted, we were not able to find parameters and training configurations where a good policy was guaranteed. A possible reason for this could be the simplicity of the weight representation. The value function approximation operating with a float vector gives the policy limited possibility for adaption. Neural networks has shown promising results for similar problems, and implementing a weight representation with this could improve the stability and the performance of the system drastically.

6.3.2 Eurobot project

As expected, building the robot was a difficult and time consuming task. For teams taking over the project I would recommend that this years robot should be used as a base. With experience from the competition I would also recommend that building the second robot should be prioritized. The smaller robot does not have to be very complex and can increase the score potential drastically. Experience from the competition show that the global positioning system was time consuming, and not really needed. None of the other teams at the competition utilized such a system.

Bibliography

[1] Thomas Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[2] Mark Boddy and Thomas L Dean. 'Deliberation scheduling for problem solving in time-constrained environments'. In: *Artificial Intelligence* 67.2 (1994), pp. 245–285.

[3] Rodney Brooks et al. 'A robust layered control system for a mobile robot'. In: *Robotics and Automation, IEEE Journal of* 2.1 (1986), pp. 14–23.

[4] Marco Dorigo and LM Gambardella. 'Ant-Q: A reinforcement learning approach to the traveling salesman problem'. In: *Proceedings of ML-95, Twelfth Intern. Conf. on Machine Learning*. 2014, pp. 252–260.

[5] Kai Olav Ellefsen. 'Dynamic Scheduling for Autonomous Robotics'. In: (2010).

[6] Eurobot. *Robomovies*. http://www.eurobot.org/attachments/article/31/E2015_Rules_EU_EN_final.pdf. Accessed on 2015-7-10. 2015.

[7] Camelia-Mihale Pintea Gabriela Erban. *Heuristics and Learning Approaches for Solving The Travelling Salesman Problem*. http://www.cs.ubbcluj.ro/~studia-i/2004-2/3-SerbanPintea.pdf. [Online; accessed 19-July-2015]. 2004.

[8] Alain Girault, Bilung Lee, Edward Lee et al. 'Hierarchical finite state machines with multiple concurrency models'. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 18.6 (1999), pp. 742–760.

[9] John Grefenstette et al. 'Genetic algorithms for the traveling salesman problem'. In: *Proceedings of the first International Conference on Genetic Algorithms and their Applications*. Lawrence Erlbaum, New Jersey (160-168). 1985, pp. 160–168.

[10] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. " O'Reilly Media, Inc.", 2013.

[11] Peter Hohnloser. 'Design of a general framework for synchronizing behaviors in a complex robot'. In: (2012).

[12] Yong K Hwang and Narendra Ahuja. 'A potential field approach to path planning'. In: *Robotics and Automation, IEEE Transactions on* 8.1 (1992), pp. 23–32.

[13] Leslie Pack Kaelbling, Michael L Littman and Anthony R Cassandra. 'Planning and acting in partially observable stochastic domains'. In: *Artificial intelligence* 101.1 (1998), pp. 99–134.

[14] Leslie Pack Kaelbling, Michael L Littman and Andrew W Moore. 'Reinforcement learning: A survey'. In: *Journal of artificial intelligence research* (1996), pp. 237–285.

[15] Gilbert Laporte. 'The traveling salesman problem: An overview of exact and approximate algorithms'. In: *European Journal of Operational Research* 59.2 (1992), pp. 231–247.

[16] Gilbert Laporte. 'The vehicle routing problem: An overview of exact and approximate algorithms'. In: *European Journal of Operational Research* 59.3 (1992), pp. 345–358.

[17] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[18] David Lee and Mihalis Yannakakis. 'Principles and methods of testing finite state machines-a survey'. In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123.

[19] William S Lovejoy. 'A survey of algorithmic methods for partially observed Markov decision processes'. In: *Annals of Operations Research* 28.1 (1991), pp. 47–65.

[20] George F Luger. *Artificial intelligence: structures and strategies for complex problem solving*. Pearson education, 2005.

[21] Sridhar Mahadevan and Jonathan Connell. 'Automatic programming of behavior-based robots using reinforcement learning'. In: *Artificial intelligence* 55.2 (1992), pp. 311–365.

[22] Francisco S Melo, Sean P Meyn and M Isabel Ribeiro. 'An analysis of reinforcement learning with function approximation'. In: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, pp. 664–671.

[23] Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2012.

[24] Volodymyr Mnih et al. 'Playing atari with deep reinforcement learning'. In: *arXiv preprint arXiv:1312.5602* (2013).

[25] Robin Murphy. *Introduction to AI robotics*. MIT press, 2000.

[26] Alex Nash et al. 'Theta^*: Any-Angle Path Planning on Grids'. In: *Proceedings of the National Conference on Artificial Intelligence*. Vol. 22. 2. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. 2007, p. 1177.

[27] Jeff Orkin. 'Applying goal-oriented action planning to games'. In: *AI Game Programming Wisdom* 2.2004 (2004), pp. 217–227.

[28] Andre Kramer Orten. 'Navigation system'. MA thesis. Norway: University of Oslo, 2015.

[29] David L Poole and Alan K Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[30] William D Smart and Leslie Pack Kaelbling. 'Effective reinforcement learning for mobile robots'. In: *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*. Vol. 4. IEEE. 2002, pp. 3404–3410.

[31] Peter Stone, Richard S Sutton and Gregory Kuhlmann. 'Reinforcement learning for robocup soccer keepaway'. In: *Adaptive Behavior* 13.3 (2005), pp. 165–188.

[32] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[33] Gerald Tesauro. 'Temporal difference learning and TD-Gammon'. In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[34] Paolo Toth and Daniele Vigo. *The vehicle routing problem*. Society for Industrial and Applied Mathematics, 2001.

[35] Nikolaos Tziortziotis, Konstantinos Tziortziotis and Konstantinos Blekas. 'Play Ms. Pac-Man using an advanced reinforcement learning agent'. In: *Artificial Intelligence: Methods and Applications*. Springer, 2014, pp. 71–83.

[36] Christopher JCH Watkins and Peter Dayan. 'Q-learning'. In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[37] H Xuan et al. 'Solving the Traveling Salesman Problem with Ant Colony Optimization: A Revisit and New Efficient Algorithms'. In: *REV Journal on Electronics and Communications* 2.3-4 (2012), pp. 121–129.

[38] Xiaoping Yun and Ko-Cheng Tan. 'A wall-following method for escaping local minima in potential field based motion planning'. In: *Advanced Robotics, 1997. ICAR'97. Proceedings., 8th International Conference on*. IEEE. 1997, pp. 421–426.