

AI in the Predator/Prey Domain

Department of Informatics and Mathematical Modelling

Frej Laursen Würtz, s011444

Supervisor:
Associate Professor Thomas Bolander

May 28, 2008

Technical University of Denmark
Kgs. Lyngby

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

This thesis concerns Artificial Intelligence in the Predator/Prey Domain. The Predator/Prey Domain is in this setup implemented as a game of catch where actual robots, in the form of Lego MindStorms, will cooperate on capturing a third part robot, either controlled by a human player or by a computer.

The concept is implemented as a Multi-Agent System (MAS), where each agent represents different strategies (AIs) on how to solve the problem. Each agent will continuously during the game negotiate on which strategy to follow for a number of iterations, and at the end of each run evaluate on the process and establish trust relations based on the progress. The MAS is further implemented in such a way that new strategies can be presented at run time.

The entire concept is handled both in physical and simulated environments.

The problem is first solved in a five times five grid with no barriers and is afterwards considered for an actual maze with internal barriers.

Preface

This report documents the aspects of developing an Artificial Intelligence environment within the Predator/Prey Domain. This report constitutes my Master Thesis as Engineer in Computer Science and is written at the Department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU).

The thesis deals with aspects of artificial intelligence and multi agent systems both in a physical environment with robots and in simulated environments. The main focus has been in the multi agent aspect where the agents cooperate in a game of catch in a maze-like environment.

The thesis is accompanied with a CD containing a digital copy of this thesis along with additional material such as program source codes and binaries and videos of game runs. The CD is best navigated through the web file *index.htm*.

The content of the CD is also available online at <http://www.flw.dk/mt>.

Lyngby, May 2008

Frej Laursen Würtz

Contents overview

This thesis strives to take the reader through the different aspects of the developing a Multi Agent System where robots play catch. The overview below serves to ease the readability of the thesis by briefly introducing the different chapters of the thesis.

The layout of report is divided into the following chapters:

Introduction which will introduce the project and the mission statement along with a prioritized list of the involved aspects followed by a problem description. The chapter also holds information on my prior experience and the prerequisites for the project.

Analysis guides the reader through the involved issues in developing the project. It is structured into the three following categories: Issues related to the environment and hardware, issues related to the Multi Agent System and issues related to Artificial Intelligence. Each section is introduced with a brief description of the containing elements.

Design and Implementation covers the most essential aspects of designing and implementing the project. It does not go into depth on all aspects, but focuses instead on a general overview, describing game runs and structures and explores some of the important agent aspects.

Testing covers the test approach for the project. It does not contain a thorough test of the project, but does instead take a theoretical approach on the aspects of testing combined with examples.

Extensions to the initial setup covers extension made to the initial setup where new mechanical parts are applied to ensure better reliability on the robots. Extensions to the maze have also been applied in the form of internal barriers, and discussions on how to adapt the existing algorithms to the new environment is handled here.

Results contains a highlight of the result made in this project. It especially concerns the effectiveness of the different strategies applied in various environments.

Further developments consider problems for future extensions to the project, especially regarding changes to the game rules.

Conclusion sums up the overall processes of the project.

Contents

1	Introduction	1
1.1	The problem description	2
1.1.1	Real life relation	2
1.2	Primary aspects of concern	3
1.3	Prior experience	4
1.4	Prerequisites	4
1.5	The content of the CD	5
2	Analysis	7
2.1	Game rules	8
2.2	The Environment and Hardware	8
2.2.1	The robot	9
2.2.2	The simulator	11
2.2.3	Findings and results	13
2.3	Multi Agent Systems	13
2.3.1	Definition of the perimeters for the MAS	13
2.3.2	Defining the agent	15
2.3.3	Agent procedure negotiation	15
2.3.4	Agent communication aspects	16
2.3.5	Findings and results	19
2.4	Artificial Intelligence	19
2.4.1	Previous attempts	20
2.4.2	Game theory	20
2.4.3	Heuristics	22
2.4.4	Winning strategy in a turn based game	26
2.4.5	A mathematical simulator	28
2.4.6	Findings and results	28

3	Design and Implementation	31
3.1	The design of the concept	31
3.2	The structure of the implementation	33
3.3	The Agent	33
3.3.1	Agent Management methods	34
3.3.2	Loading of procedure	35
3.3.3	Communication	36
3.3.4	Implementation of the algorithms	38
3.4	Findings and results	39
4	Testing	41
4.1	Ways of testing	41
4.2	Structural testing	42
4.3	Functional testing	43
4.4	Usability testing	43
4.5	Findings and results	44
5	Extensions to the initial setup	45
5.1	Extensions to the mechanical setup	45
5.1.1	Discussion on the robot	46
5.1.2	The color sensors	46
5.1.3	The improved robot algorithm	47
5.1.4	Findings and results	49
5.2	Extension to the game setup	49
5.2.1	Defining the problem	50
5.2.2	Heuristics - Reapplied	50
5.2.3	Game theory - Reapplied	51
5.3	Findings and results	53
6	Results	55
6.1	The heuristics	55
6.2	The winning strategy	57
6.3	Agent negotiation and procedure swapping	58
6.4	The robot	58
7	Further developments	59
7.1	Changing the game rules	59

8 Conclusion and perspectives	61
References	65
A Pseudo code for the Heuristic	67
B Pseudo code for the Winning Algorithm in a Turn based game	69
C Pseudo code for the NXT routine	71
D Unit Test of the Winning Strategy	73

Chapter 1

Introduction

Artificial intelligence and robots are subjects which often get the attention in modern science. Trains should drive without train drivers, cars must find their way through traffic on their own. Robots assist in medical operations and computers challenge the human mind in diagnostics, chess, computer games and endeavor to excel in every practice where humans excel.

These concerns are just some examples of why Artificial Intelligence and robots get so much attention in modern science, which also forms the basic for my choice of Master Thesis.

As an outline, I wanted to do something challenging, and something fun. The IMARS lab at IMM provided the possibility to combine robots and AI without spending a huge amount of time on automation which is often the case when working with robots. This would allow me to focus on the other aspects of combining AI and robots.

After some discussion on how to combine these two aspects into a reasonable project – and greatly inspired by the old PacMan game – I came out with the following Mission Statement:

I want to build a game where robots play catch.

This is a pretty simple statement, thus nicely suited as the basic outline for the project. This of course needed to be concretized and elaborated, and I further wanted to include a multi agent aspect. The statement was thus rephrased into:

Let a number of robots be individual agents (predators), which will collaborate on capturing a single robot (prey).

The intention behind this statement is to create a domain where a number of robots act as individual autonomous agents, which given a common goal will cooperate on achieving this goal. In this case capturing a prey robot. This predator/prey capturing scenario is often referred to as the predator/prey domain.

In the rest of this chapter I give a precise problem description and relate it to a real life scenario, and afterwards define the essential aspects of the project. Given the above statement there are plenty of issues to be explored but not enough time to explore them all. I will therefore try to pin down the involved aspect and prioritise these.

After this, I will briefly introduce my prior experience and the prerequisites for the project.

1.1 The problem description

With regard to the mission statement, the following constitutes the problem description.

It is required to develop a setup where robots are capable of navigating a maze in a convincing manner. The robots should be implemented as agents, which will interact with each other in a game of catch, where a number of agents will play the role of predators, and a third party agent will act as the prey. The predator agents should be capable of capturing the prey. This should also be the case, when the prey is controlled by a human player.

This setup must be implemented both as a virtual (simulated) and a physical environment.

The following should further comply:

The agents should be individual autonomous entities. Meaning that there must not exist some master agent which will control or dictate the actions of the other agents. The agents should furthermore be able to adapt to new procedures/approaches at run time, meaning that it will not be necessary to take down the system in order to update it or make it able to cope with new situations.

The last requirement is introduced to make the MAS implementation dynamical and make the framework applicable to other domains, as when applied in a scenario as described in the real life relation in the following.

1.1.1 Real life relation

In today's high complexity productions, more than one part (company) is often involved in the development process. The product might be divided into different sub-products or outsourced/contracted to different companies, which then develop their subproduct. Most parties are often reluctant if not completely unwilling to share their knowledge or their area of expertise in order to preserve their industrial advantage.

In this project case, each agent can be thought of as a representative of a contracted company, who will not relinquish any knowledge to the other companies, but they still cooperate in order to achieve the common goal. The company can design the agent to do their part of the shared process, and the agent will then interact on behalf of the company in the network and coordinate the work for the company specific part of the process.

1.2 Primary aspects of concern

There exist a huge number of implementations, and a huge number of aspects to explore, and it is entirely impossible to explore them all. I will therefore try to list those, that I see as the most obvious and explain why I have chosen to either dismiss or pursue that aspect. Of these Multi Agent Systems and Artificial Intelligence will be given the most focus.

As the project involves robots, **hardware** and **automation** plays a natural role. I will however only treat this aspect as the most basic need, and only pursue this as far as is necessary to get a working setup. A lot of additional effort can be put into e.g. sensors or odometrics, to make the robots drive more precisely and minimize the risk of getting out of course. I do however think that other aspects have more theoretical value and will pursue those instead.

In order to further limit the time spent on the physical aspects of the robot, I will develop a **simulator** which shall be able to mirror a perfect physical robot and the environment in all aspects related to this domain. By perfect meaning a robot that does not suffer from errors related to mechanical or sensor related issues. This allows me to perform a lot of testing, both actual and conceptual without being restrained by the speed and inaccuracy of the robots

The setup should furthermore be a **Multi Agent System (MAS)**. I have chosen this in order to make the project more contemporary, and since MAS provides a means for the high dynamism, as required in the problem description. This area will be given the most focus throughout the project. A more refined definition of the multi agent part will follow in Section [2.3.1](#).

The entire purpose in the mission statement is to make robots cooperate on capturing a prey, (i.e. third party robot). **Artificial Intelligence** therefore plays a significant role in enabling the robots to do so. I will in this respect discuss different approaches within this field in regard to capturing the prey in an efficient manner.

Since the robots will cooperate, they need to perform some kind of **communication**, which in implementation aspects will involve measures regarding choice and implementation, of transport protocols. These aspects will only be handled theoretically, since implementation methods are plentiful and well documented, and would not add anything new to the project. In this regard, a number of solutions will be outlined.

The choices made, are based on importance related to the mission statement and on personal interest. I will however stress, that it has been necessary to make these decisions since it is impossible to contain all aspects within the size of the thesis.

1.3 Prior experience

My prior experience on this subject primarily concerns distributed systems, both multi agent and client/server systems. Besides this I worked a great deal on security in networks, but this will be put to little use in this project. My knowledge on AI is primarily limited to a number of algorithms regarding sorting and searching.

1.4 Prerequisites

The IMARS¹ lab at IMM forms the basic for the physical setup. IMARS is ideal for the setup at hands, since it provides a number of highly customizable robots through the Lego MindStorms concept². This allows me to construct a robot suitable to my needs without spending much time on hardware related concerns.

I am further able to benefit from the knowledge earlier acquired in the lab by Christian Agerbeck [1] and Anders Lemke et al [5] who did a great deal of ground work on interfacing with the robots and making them act in a physical domain similar to mine.

The first group, Anders Lemke et al, who were the pioneers in this lab made a domain where robots would navigate around in a grid layout, and pick up objects defined by their color. The robots would only be allowed to pick up object with their given color, but were able to communicate the location of objects with different color to the other robots.

While making the project, the group developed a C# wrapper for communication with the robots from a PC, along with some subroutines for making the robots perform simple commands and navigate in the physical environment. Both the wrapper and the subroutines have been used as a basic in this project.

Christian Agerbeck later made a predator/prey implementation based on the previous work, where the agents were designed as BDI³ agents. Predators would first try to locate the prey and afterwards negotiate for positions adjacent to the prey. This approach unfortunately gave rise to situations, where a predator might have to move around other predators to get to the negotiated position, which in some cases led to a unnecessary high number of moves for capturing the prey.

With the knowledge of these two reports, I have tried to benefit from their experience in the IMARS lab, and not waste too much time on documented shortcomings. Other shortcomings do however exist, and some of the issues from Agerbecks project form the basic for discussion on the domain, as will become clear during the report.

¹IMARS is a lab containing Lego MindStorm robots and accessories

²Lego MindStorms is basically a means to construct robots through building block (like in the normal Lego concept).

³BDI is short for Belief-Desire-Intention

1.5 The content of the CD

The report is accompanied by a CD, where a digital copy of the report can be found along with additional material, which includes the program code for the project. Both as sources, source documentation and binaries. The CD also contains videos of game runs in both the physical and virtual environment.

The CD is best navigated through the index.htm file in the root directory, which also explains how to run the program. The CD also contains user manuals for the program.

The content of the CD is also available online at <http://www.flw.dk/mt>.

Chapter 2

Analysis

As introduced previously, this project contains more aspects than it is possible to cover within the scope of the thesis. I will in the following sections perform an analysis of the mentioned aspects, and go into depths with those I find to be the most relevant for the chosen solution.

As stated in the mission statement, the primary goal of this project is to perform a game where robots play catch, or more formally known as *the predator/prey domain*. This is defined as a setup where a number of predators chase or eat a number of preys. The domain is usually used to model the behavior or life cycle of different animals. For instance foxes and rabbits in different natural environments or fish in the ocean. Most often in these setups, constraints are based on the dimensions of the environment, the growth and number of species and their speed. These uses are however not applicable in my case, since I am limited by other physical constraints, and that my scenario will concern only one prey and a number of predators. This does unfortunately make a lot of existing literature unsuited for my needs.

A brief overview

For the analysis of this project, I will first define the domain and then analyze the available means, and hereby discuss the problems related to hardware, automation and other physical constraints. I will discuss the benefits of a simulator in this part as well. Afterwards, I will go into detail on the MAS part and further concretize the project in these aspects, since this part is considered the core part of the project, and is intended as a proof of concept. The MAS section contains discussion on negotiation, trust and communication.

Finally I will go into discussion on artificial intelligence, where I will discuss three algorithms for solving the problem: Game theory, heuristics and a winning strategy in a turn based approximation.

But let us first setup the game rules for the scenario at hand.

2.1 Game rules

The game rules are very simple. The predators goal is to capture the prey, whereas the preys objective is to evade the predators.

The prey will move around at random in the initial state, but will in the final state be controlled by a human player. It is a requirement that the prey will move around. It might chose not to perform a move, but it must move eventually.

Capture is defined as the situation where the prey can no longer move. As if the prey is cloistered up against a wall, or totally surrounded by the predators. An illustration of captivity is shown on Figure 2.1

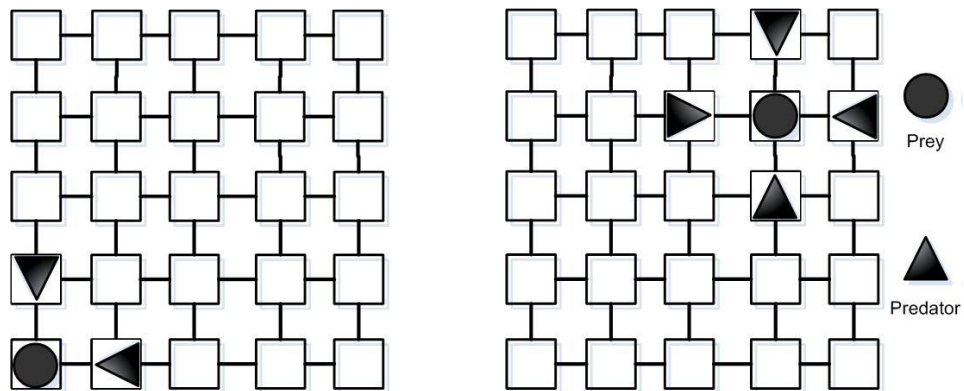


Figure 2.1: In the first figure, the prey is captured by being cloistered by walls, and in the second figure, the prey is captured by being completely surrounded by predators

The prey cannot be captured by less than two predators, since one predator cannot trap the prey in such a way that the prey can no longer move, and the predators are therefore required to communicate and work together in order to win. This requirement is made both in order to solve the physical problem of occupying a position by more than one robot, but also in order to add the multi agent aspect to the problem.

It is the ambition always to make the predator agents succeed in capturing the prey, even when it is controlled by a human.

2.2 The Environment and Hardware

As introduced earlier, the IMARS lab and the Lego Mindstorms robots (see Section 2.2.1) form the basic for the hardware and the physical environment.

The environment is primarily based on the setup used by the two previous projects, where a number of Lego road plates form the layout of the maze. Initially, it only forms a five times five grid, but it is intended to include barriers at a later development iteration. Figure 2.2 shows the maze in its initial layout.

As the figure shows, the paths which the robots can travel are marked up by black tape, and the intersections are indicated by silver colored tape. This is done in order to ease the recognition by the robot sensors.

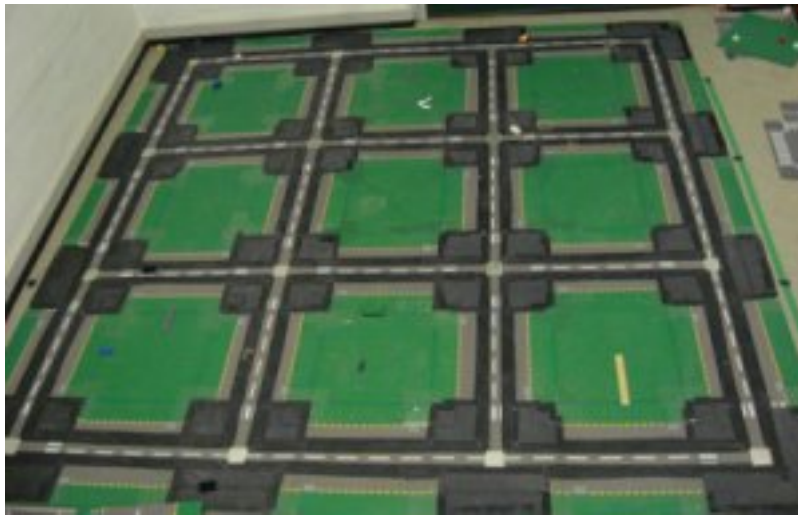


Figure 2.2: The maze in its initial state, though only in a four times four grid.

As the viewer might observe, the maze is in this state very simple. This is again done in order to ease the development of the other aspects of the project. It is however kept in mind during the development process that the maze will become an actual maze in a later iteration (see Chapter 5), and that the maze might be of a bigger size than just five times five¹.

In the following, I will describe the nature of the robot, and the benefits and requirements for a simulator.

2.2.1 The robot

The hardware constitutes the actual robot, its mechanicals, sensors and the NXT brick.

The robot is made up by Lego Mindstorms bricks, and is very much based on the design used in the previous two projects, which is why I will only briefly introduce the Mindstorms concept. For more information, please refer to [1] and [5].

The Lego Mindstorms concept is a basis for building small robots through Lego building blocks. The actual robot capabilities are contained in the NXT bricks, which basically is a small computer attached with three output ports for controlling motors and four input ports for receiving information from a variety of sensors. The NXT further has the ability to communicate with other devices through bluetooth.

Along with the design of the robot, the previous groups developed a number of sub-routines for making the robot act, like following lines, turn and communicate. These routines have been modified to suit my needs. One important note on the NXT brick, is that its computational power and memory is very limited, and it is therefore necessary to handle these issues on an external device.

¹The five times five size is a restriction by the physical size of the lab.



Figure 2.3: Picture of the robot

Though the end-design (Figure 2.3) is very similar to that of the two previous projects, measures have been taken in the attempt to eliminate unsolved shortcomings. The most important of the shortcomings, are the sight of the robot, which will be handled in the following. Other shortcomings relate to the speed of the robot, its ability to follow lines properly and its sensitivity to roughness in the surface. These concerns are however reassessed in Chapter 5

2.2.1.1 Problems with the sight of the robot

In the initial stage, a lot of probing and testing was done in order to eliminate the problem from Agerbecks [1] solution, where the robots moved in hacking motions. The reason for this was that the robots needed to check if the prey was in front of it, which it would do by the ultra sound sensor. The problem arises, since only one robot can use the sensor at a time, because it would otherwise create false values for the others, since the sensors all operate on the same frequency. The robots then needed to stop up every few steps and take a look.

I tried to solve this problem by letting the robots “look” concurrently while moving, which would remove the hacking motion. This worked well with two robots, but already with three robots, the robots did not receive permission to look before it reached the next intersection, which caused the robots to crash into each other. This

is mainly due to the necessary delay before making a new ultra sonic reading² This could be solved by slowing down the robots, but since they already move quite slowly, this was not an option.

Some other solution could involve light sensors, where the prey emits a light beam. The predators would then be able to pick up the light emitted and based on the intensity pinpoint the direction of the prey. These readings would by two predators (or more) enable them to calculate the prey's position based on orientation vectors and the predator's positions. The problem with light sensors is however that they are very sensible to variations, e.g. whether its a bright day, if the room light is turned on and so on. But given more accurate sensors than what I currently have available, the approach might work.

I have instead solved the problem by making the following assumption: the prey will always move around in an open area which is traceable from the sky, hence the predators will always know the position of the prey. In a real hunt, it would correspond to having a helicopter track the prey.

This simplifies the initial problem, since the predators need no longer have a strategy for searching through the maze. This can however still be simulated, by first letting the predators know the position of the prey, when they are in close proximity to it. This will however be left for another project.

Since the predators no longer make use of an ultra sound sensor, it opens the possibility for the prey to use it, since it is now the only one. This can be done, but besides from the above mentioned problem, the sensor has problems recognizing the objects, and defining actual distances. Agerbeck solved this problem to some extent by developing a reflection wall, which ensured better readings. I have however chosen not to, since the prey in the end will be controlled by a human with knowledge of the predator's positions, and will not be limited to the immediate sight of the robot.

2.2.2 The simulator

As explained in the introduction, the availability of a simulator would greatly speed up the development phase. The simulator should be a representation of a perfect environment and perfect robots. Another way of viewing it, would be as an underlying layer, which keeps track of all elements within the environment and ensures that no illegal actions can be made, e.g. moving outside the boundaries of the maze. Figure 2.4 shows a simulated representation of the environment.

The simulator is however mainly intended as a development tool. I will in the following explain my thoughts on how to develop a simulator.

I found that the biggest problem about writing a simulator, was to determine how to transform the physical world into a mathematical abstract world. When I was going through the options, I had a lot of different ideas on how to build a simulator that was as general as possible from the beginning, which would allow me to make adjustments to the physical world which could immediately be handled within the simulator as

²Agerbeck [1] made experiments on how long to wait, before making a new reading, in order to remove any residues from the previous reading.

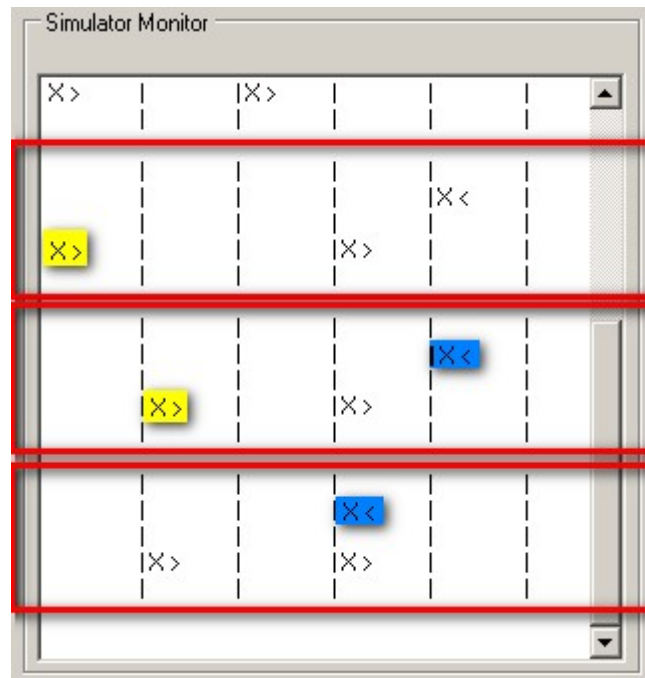


Figure 2.4: An example of a simulated representation of the environment. The 'X' represents a robot and adjacent arrows indicate their orientation. For every move, the simulator prints out a snapshot of the maze, which is indicated by the boxes. The highlights indicate which robot performs the move.

well. This led me to thoughts about complex graphs, where each node would allow an arbitrarily number of connections, which again let me to discussions on how to make sure that these connections were actually possible in the physical world. In the end, I ended up with the following concept:

Construct a simulator which is only able to model the current environment, but create it as an isolated instance, so that it can easily be replaced with another if necessary.

By isolating the simulator, I further get the advantage that changes made in other parts of the program do not directly reflect in the simulator, and thereby, that errors introduced elsewhere will be caught by the simulator.

The simulator was initially designed to simulate the ultrasound sensor as well, so that the simulator would be able to rise interrupts in the event that the prey had been spotted. This setup was later discarded.

So as it is, the abstract representation of the physical maze is a simple coordinate system, where the virtual robots can move around within the boundaries of the coordinate system.

2.2.3 Findings and results

Though the robots are made highly customizable through the Mindstorms concepts, they unfortunately suffer from a lot of problems regarding the reliability. As for instance when following a line or regarding the sight. These problems have however not been of primary concern, as instead I have relied greatly upon earlier discoveries on the subject. I do however reassess these problems in Chapter 5. As for the sight issues, I have circumvented it by letting the prey broadcast its position.

To further reduce the dependence upon the robot's uncertainties, a simulator has been developed, which should mainly be viewed as a tool for trying out ideas and concepts. It is in its end design fairly simple since a lot of approaches were discarded along the way. It has however been most beneficial.

2.3 Multi Agent Systems

The Multi Agent System (MAS) is included to make the project more interesting and more contemporary. The multi-agent approach is often used for solving complex problems or dividing huge products into smaller bits and let agents work on these individually, and then concatenate the results into a complete solution. It is often combined with distribution of resources, where SETI@Home³ is probably the most known example, where idle computer power on home PCs worldwide are used to analyze signals from outer space in the search for foreign intelligence.

The intention behind the MAS approach for this project is however not to search for intelligent life or solving complex problems, but instead to make an intelligent and adaptive system. It is the intention that the system should be intelligent in the sense that it can easily adapt to new approaches simply by supplying a new agent. The system will in this way always be as intelligent as the most intelligent agent.

I have found the primary elements of concern in this area to be cooperation between the agents (negotiation), trust issues and inter-agent-communication, which will be discussed on this section⁴. But before going into these issues, let us first define the perimeters for the MAS

2.3.1 Definition of the perimeters for the MAS

The MAS in this content is intended to consist of distinct entities, where there exists no master intelligence which will dictate the other agent's actions. The agents must furthermore be able to adapt new procedures/approaches on how to achieve the goal at runtime, meaning no downtime is necessary in order to add new procedures.

This is meant to be archived, by letting each agent represent a given approach (algorithm/intelligence) for achieving the goal. The agents will then negotiate on how to proceed by evaluating the different approaches and then decide on a procedure,

³For more information about SETI@HOME see [9].

⁴A lot of other issues can be address when designing multi-agent systems and different approaches and concerns are discussed in MultiAgent Systems by Michael Wooldridge [11].

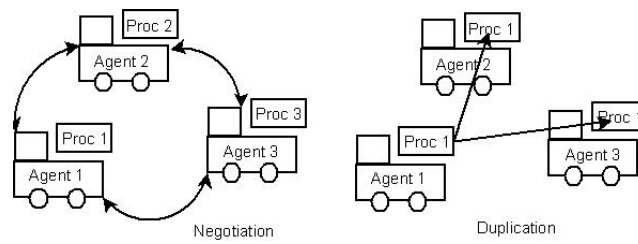


Figure 2.5: When an agent has won the negotiation its procedure on how to solve the problem is duplicated onto the other agents in the network

which will then be the active procedure for a given number of iterations or an interval. The procedure will then be copied onto the other agents and executed for the defined interval (see Figure 2.5).

At the end of each iteration, the agents will then evaluate on how the process went, and on the basis of this, establish trust relations to the agent suggesting the procedure. The trust value will be given a negative value in the event that the procedure did not accomplish the alleged achievement, and hence the agent providing the procedure will be less likely to be selected for the next approach. Figure 2.6 illustrates the approach by a state diagram.

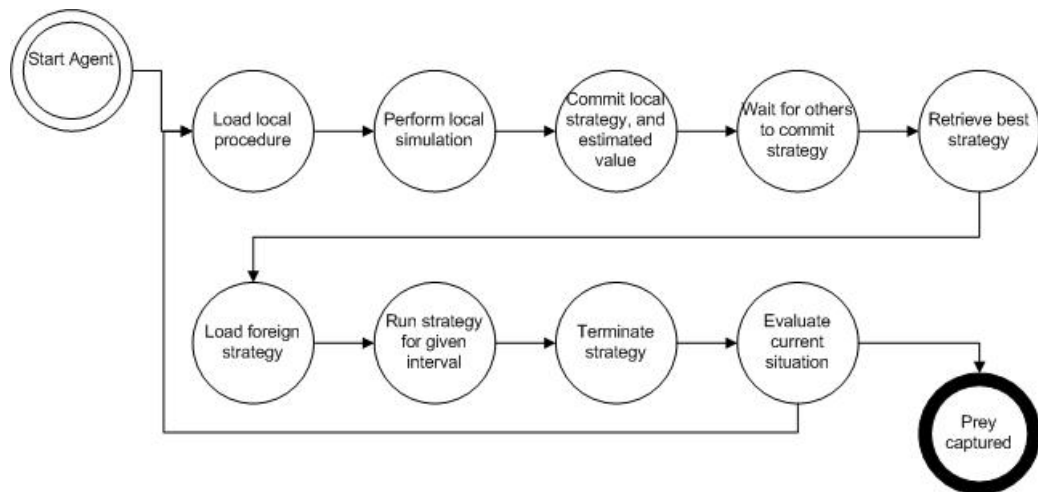


Figure 2.6: Game run state diagram of the agent procedure

Updates to the system is done simply by supplying a new agent with the new algorithm to the system, which through the negotiation process will be propagated to all agents in the network.

It is the intention that the MAS requirements should yield a system which should be viewed as a proof of concept, where this approach could be applied to any system where the success criteria for the domain can be evaluated objectively. The proof of concept is in this project applied to the predator/prey domain.

It is worth noticing that some of the traditional reasons for choosing MAS implementation are to share resources and combine results. This is not the case in this

setup, which instead has a great overhead on resources. Compared to a master/slave relation, which would be the alternative, where one agent wins the negotiation and afterwards dictates the behavior of the other agents the benefits are, that the MAS approach is a lot more fault tolerant, since it will be able to continue operations even if a number of agents go down. Consider for instance the following example:

Assuming that all agents still contain different procedures/strategies on how to solve a problem, the agents will still negotiate for the best procedure and agree on which procedure to follow. If the winning agent in this case would establish a master/slave relation instead of duplicating itself onto the other agents, and then break down before the process was complete, everything would be lost, and the other agents will not be capable of continuing, since the master was the only one with the necessary information. If the procedures are only intended to run for short intervals, the loss will be minimal, but if the process were to run for several days, a lot of resources would have been wasted. As opposed to this, the duplication approach will ensure that every agent knows the whole process and will always be capable of continuing as long as a sufficient number of agents exist in the network.

2.3.2 Defining the agent

The agent in this concept can be thought of as a self aware robot. It knows where it is in the domain and it knows its objectives. It further knows what other agents there are in the domain and can communicate with these. The agent does however not know how to achieve its goal, and it does not know how to operate the mechanical parts of the robot. For this it needs an additional intelligent procedure or algorithms which will be discussed in Section 2.4.

The agent is able to communicate with the other agents in the network, and to relay information between the intelligent procedures running on the agents. It is further responsible for performing the negotiation of which procedure/intelligence to run, and to load and terminate these. And finally to evaluate on the success of the approach.

2.3.3 Agent procedure negotiation

The negotiation process arises when the agents in the network shall decide on which procedure to follow. Since each agent represents a given approach on how to solve the problem, the agents need to agree on which approach to follow. It is necessary to agree on a common approach in order to ensure cooperation. If each agent uses its own approach, the system would be totally autonomous, and no coordination or cooperation would be possible. The negotiation should further be based on objective factors, i.e. something that can be observed from the physical domain, and not some subjective calculation performed by the agent's local procedure. These factors would naturally be closely related to the goal of the domain, which in this case is to capture the prey. Success criterias could therefore be based on the time it would take to capture the prey, by a given approach. This might however be infeasible in large domains, or with some AIs, e.g. huge game trees or if different approaches would be better at different stages of the game. Here the overall distance from all predators to

the prey after a given interval might be better. Again another approach could be to make the estimate based on the number of options the prey has for moving. In this sense, a situation where the prey is cloistered by a barrier would be better than if the prey can move in all directions. For this project, however, it does suffice to base the success evaluation on the overall number of intersections that the predators need to pass in order to capture the prey.

Issues on communication between the agents are handled in Section 2.3.4.

2.3.3.1 Agent negotiation trust issues

Since the estimation on the effectiveness of an approach is performed by the local intelligence, the ability to calculate these estimates might not be evenly good, and the local intelligence might not be reliable in its estimation. This might either be intentional or unintentional. It is therefore necessary to consider some sort of trust, which will enable the agent to have some means of evaluation whether a procedure can make a reasonable estimate for achieving the goal. A way to handle this, would be to introduce trust values. When a procedure wins the negotiation and is loaded onto all the agents in the network, it will only run for a given interval. This can either be its own estimate of how many moves is required to reach the goal, or it can be a max number of runs (or a time limit, if the procedure simply makes the agent stand still). The procedure would afterwards be evaluated on its perfectiveness. This should of course be based on the same criterias as for evaluation of the effectiveness during negotiation. If the procedure fails it would be assigned a negative value. This strives to ensure that fallible procedures will not render the system unusable by always winning the negotiation and not achieving the goal. By introducing a trust value for use for future negotiation, it is ensured that fallible procedures will be sorted out over time and other agents will get their pick as well.

This approach also allows for assigning positive trust values, hence make a proved procedure more likely to be chosen in the future as well. Some considerations do however need to be taken into account. If the system has been running for a while, and a given procedure has been favored and a new procedure is introduced into the system, the old procedure might still be chosen in the negotiation, because it is proved, even though the new procedure might be better. Both for the negative and the positive trust values, it might be feasible to reset the values at intervals or let the values diminish over time.

2.3.4 Agent communication aspects

As with every other MAS, there need to exist some form of communication between the agents, and a number of approaches exist for doing so. I will in the following describe the involved concerns.

Since the theoretical standpoint is that each agent should be totally independent, the ideal form of communication would be to let the agents interact in a peer-to-peer like manner, where each agent would communicate with a number of peers, which again would communicate with another set of peers, and thus in the end, all peers would

have been reached. This would work, and would eliminate any single point of failure, but it would be highly inefficient in measure of time, since a lot of transactions would be made to agents, which had already received the message, and it would be difficult to determine if the received message was actually the message for the last move, thus there could never exist any exact knowledge of all the positions at any given time, since the peer to peer network is asynchronous.

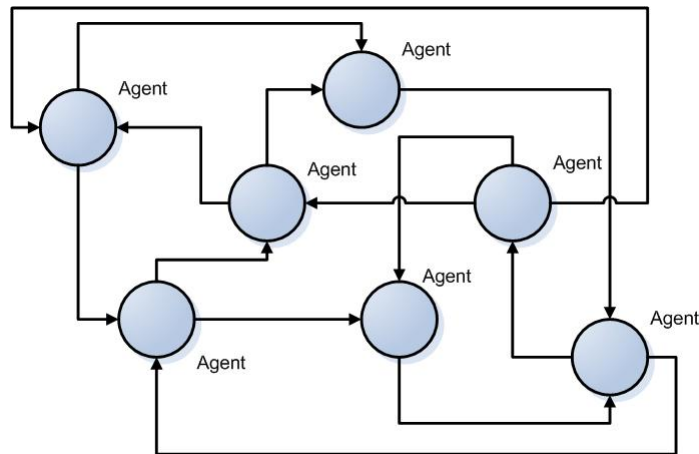


Figure 2.7: Peer-to-Peer communication diagram

Another approach would be to let each agent contain a list of all known agents, and then for each transmission ensure that every other agent had received the message. This approach is more suited for the current needs, but an important question still exists. How should the list be maintained? How should new agents be added to the list, and how should broken agents be removed from the list?

Let us assume that the communication protocol is settled. We then need some sort of discovery service. A feasible approach could be to let each agent broadcast a “I’m alive” signal at given intervals. This would allow new agents to pick up the signal, and request enrollment into the list of known agents in the network, and it would allow existing agents to discover new agents. It is then important that the list is updated at each agent when a new agent is enrolled, in order to ensure that all agents act on a correct set of environment variables. This could further be ensured by introducing a broadcast signal, that would halt any given process until all lists are updated.

The “I’m alive” signal could further be used for removing broken agents. If an agent no longer broadcasts the signal, the other agents can send a form of acknowledge request to ensure that the agent is actually broken, and if the acknowledge request is not confirmed, the agent can be removed from the list by the mentioned update signal.

This would be the preferred approach for this implementation. But let us just consider a simplification. The idea is to introduce a master station or relay, meaning a central point of communication, where each agent reports to and retrieves its information. The big downside of this is that this introduces “a single point of failure”, meaning that if the master station brakes down, the entire network brakes down. The benefit of this is, however, that this allows for a control point outside of the actual network,

which can respond to human interaction. E.g. if an agent needs to be removed from, or inserted directly to, the network, or if some kind of surveillance of the network is needed.

It further has the advantage that it allows for simplification of the communication, since the master station will have a static address, which new agents can report to, and messages only need to be transmitted n times (where n is the number of agents) and not n^2 times (if each agent needs to communicate directly with each of the other agents). This is, however, not entirely true, since a number of procedures exist to limit the number of messages. The token ring approach for instance, where messages can be viewed as a train with goods, which travels in a circle around to all connected entities. New messages are appended to the existing train and delivered when the train arrives at the destination of the message. Some fail safe mechanism though has to be implemented in order to ensure correct delivery, if for instance the chain brakes.

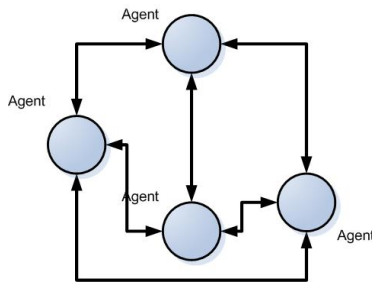


Figure 2.8: Communication where each agent communicates with every other agent

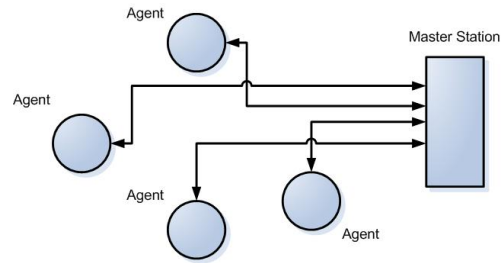


Figure 2.9: Communication with a master station for registering agents

In the decision of which approach to take, it is worth noticing that the master station approach is considerably easier to implement, and is thus more suited as a first approach.

Some measures can be taken to improve the failsafe of this approach. Firstly a redundant implementation of the master station could allow for a backup master station to take over if the first fails, and secondly a backup procedure could be implemented in the agents, where these would revert back to the first mentioned approach without a master station, for as long as the master station is down. It should, however, be noticed, that the master station is considered to be a lot more stable than the agents, and that the probability that the master station will malfunction is much lower than the probability that an agent will malfunction. This is due to the fact that the master station is an isolated instance which merely serves as a control point and communication relay, and does not interact directly with the physical environment.

The implementation of the above is pretty straightforward but time consuming. I have earlier made a peer-to-peer implementation in JXTA and a client/server implementation over a subset of HTTP 1.1⁵. A lot of other approaches and implementation strategies are given by Kurose and Ross [4] and Robin Sharps book [10] on protocol design goes into more complex details.

⁵These implementations, among others, are made in the IMM courses “Parallel Systems (02220)” and “Distributed Systems (02222)”

2.3.5 Findings and results

Multi Agent Systems can be implemented in a lot of different ways, and there is a huge number of different problems one can dwell on. I do, however, find that I have gone through the most relevant in regard to the current setup, namely negotiation, communication and trust relations.

For the negotiation part, I have found a negotiation based merely on absolute estimates of the necessary number of moves for capturing the prey to be sufficient for the initial implementation. For more complex implementations, more relative estimates should, however, be used. As for the trust relations, these will ensure that fallible agents will be less likely to be chosen in the future, and thus ensuring over time that the most beneficial procedure will be selected at all times. Finally, communications in the form of a master relay, has been chosen for this setup, since it is the most beneficial for an initial setup, whereas peer-to-peer-like communication forms should be considered for more extended setups.

Given more time, the most interesting problems to explore further would probably be to make the MAS more robust and dive into different means of making the MAS able to solve problems, where agents break down, time out or act irrationally. It would further be interesting to include misbehaving agents, and then make an analysis on the security aspects of the system.

2.4 Artificial Intelligence

Artificial intelligence in this respect concerns the methods for capturing the prey. It is indented as an isolated procedure which can be switched out at run time. The procedure can be regarded as the brain of the robot, and is responsible for coordinating and planning the moves.

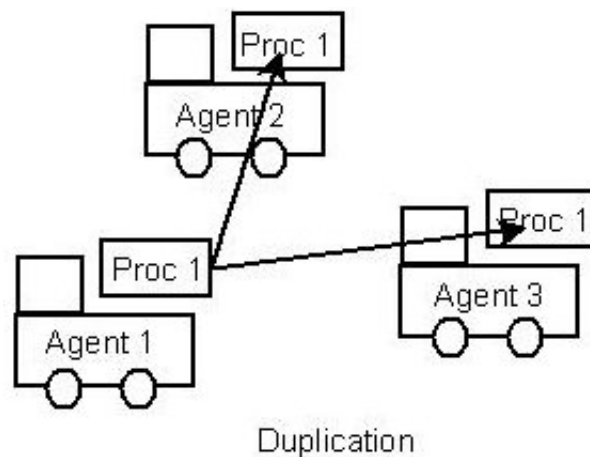


Figure 2.10: When a procedure is chosen as the superior, it is copied onto the other agents in the network. In this case, the procedure of agent 1 is duplicated onto the others

In order to achieve the goal of capturing the prey, a number of approaches within the field of artificial intelligence were considered. Especially game theory and heuristics.

are autonomous. But because all agents move with the same speed and the distance between all intersections are identical, it was presumed that all agents would have time to perform one move, before any agent could perform the next move. This approach is, however, fallible as will become clear later in this section. As for now we will pretend that it works.

The idea behind game theory, when applied to this setup, is to regard the domain as a game where each part (predator versus prey) moves by turn as in a common board game. Every possible move for each turn of the entire game is then drawn up as a tree, as illustrated in Figure 2.12.

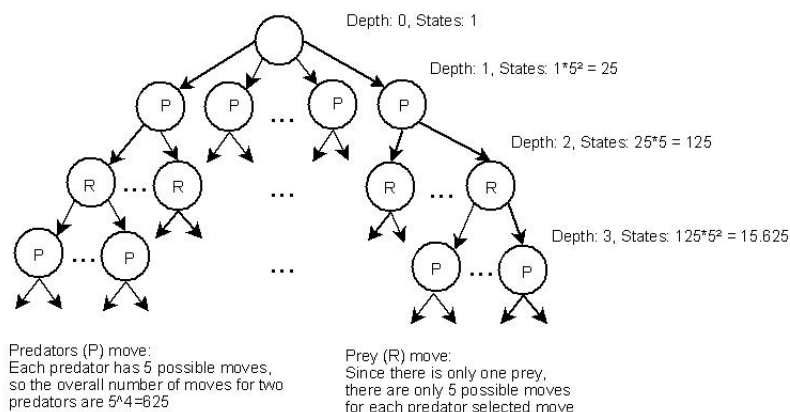


Figure 2.12: Game tree for this domain. R represents the prey and P represents the predators.

For this domain, each predator have five different options available for each turn (To move to either of the four adjacent positions or to stand still). In a setup with four predators, this gives 5^4 combinations for each predator move. The prey has the same options, so it adds a factor five for each prey turn. By this approach the game tree becomes very big very fast, and it is therefore not feasible to make a complete game tree. The process can, however, be limited by only building the game tree to some depth for each iteration, and then call a heuristic evaluation function for estimating the advantageous of the position and then perform the move based on a minimax search or through alpha-beta pruning.

The next problem arises when the predators do actually move, since situations might arise, where more than one situation yields the same value, and predators might therefore have more than one possible move to choose from in order to reach this situation. The predators then need to have an additional evaluation function to determine which situation to choose, and what position to move to. This could be done by negotiation, but it would be more beneficial, if the predator could decide this itself, since additional negotiation would prolong the process.

But given that we can limit the tree to a reasonable degree and that the predators can decide on the right situation and the specific movements, one major problem still exists.

The problem arises because of the nature of the domain. The approach would probably work in simulation, but in reality movements turn out to be highly asynchronous. Unfortunately it does not take a constant time for a predator to move from one

intersection to another even though the distance is the same. This is due to the nature of the sensors and the line following algorithm, where the agents move in sweeping motions, and might even start the movement at an odd angle. But more importantly, when a predator makes the calculation it is highly unlikely that the other predators are standing still. They are probably on the move to another intersection, whereas the prey is in constant movement, so the input on which the calculations are made, is already obsolete at the time the predator receives it. The effect of this can be minimized by synchronizing the predators, but this will in some cases allow the prey to perform two moves while the predators only perform one. One more limitation exists in the available computational power, since all agents currently run on the same computer, it is not possible to perform the necessary calculations in time.

The game theory approach should, however, not be totally disregarded since it has the valuable ability to determine whether a solution exists for a given situation. It might also still be applicable for this domain if the following constraints could be removed:

- The robots should be able to move in smooth motions from one intersection to another, meaning that issues related to the line following and turning of the robot should be resolved.
- Optimization should be made on the framework or the agents should reside on different computers in order to make the necessary calculations in time.

This does, however, not mean that the turn based approximation is inapplicable, just that it rises additional issues when applied at a larger scale. These concerns have, however, been solved to some degree by improvements discussed in Chapter 5, which is why I in Section 5.2.3 try to re-apply game theory on the domain.

But let us for now assume that the above issues are already solved, the game theory would still require a number of heuristics or evaluation functions which leads consideration on to the following approach

2.4.3 Heuristics

Heuristic functions are a means to estimate the effectiveness of a move or decision without performing the calculation to its end, as in the case where the game tree is too huge to build to its end. Common heuristic approaches are hill-climbing and simulated annealing⁶, where the search space is defined as a graph, and the best solution is defined as the global maximum. The heuristics are here methods for finding the global maximum without considering the entire search space. I will, however, consider a different approach, where the search space will be limited to the positions immediately adjacent to each predator, and the heuristics will be functions that “guesses” the advantage of each of these positions.

In conjunction with the game tree attempt, it was the idea simply to estimate the effectiveness of a single move through heuristic functions. This section will highlight

⁶See [6] for further information on these techniques.

the available options and discusses the effectiveness of these when applied in the domain.

When observing the domain, a number of criterias comes to mind for estimating the advantage of a position. The following will give an introduction to these.

Distance to prey. The overall distance to the prey from each predator would be a good indicator of the favorableness of the position, since the capture situation occurs when the predators are adjacent to the prey.

Limitation of prey options would be another approach, in positions where the prey somehow is limited in the number of available moves, i.e. if a position is blocked by a predator or if the prey is limited by the barriers of the maze. This would however not be enough on its own, since a great number of positions exist where the prey will have the same number of available moves, but it might be suitable in combination with other functions.

Distance between other predators. It will in most cases be beneficial to let the predators spread out and cover more ground. But it will be unfeasible in the end game, where the prey is almost captured.

Advantageous predator positions which in most cases are positions, which maximize the available moves for the predators or reduce the prey's available moves.

In order to make a feasible heuristic all the above elements need to be taken into account.

2.4.3.1 Predator/Prey distance function

This heuristic will as mentioned only be applied for calculations on positions immediately adjacent to each predator. This distance function therefore only considers the distance from the current predator to the prey.

$$f_{preyDist} = Weight_{preyDist} \cdot (Constant - \text{Min}(prey, position))$$

The distance in this setup can be calculated merely as the direct distance, since there are no internal barriers. The function result should then diminish as the distance increases. One implementation of this function could be as the code in Figure 2.13, where the value is reduced by 50% for each time the distance is increased by one.

4	15.00	10.00	6.67	4.44	2.96
3	10.00	6.67	4.44	2.96	1.98
2	6.67	4.44	2.96	1.98	1.32
1	4.44	2.96	1.98	1.32	0.88
0	2.96	1.98	1.32	0.88	0.59
y/x	0	1	2	3	4

Table 2.1: Table with the values of the predator/prey distance function in a 5x5 grid

```

1 Method (returns number):preyValue{
2   dist = distance(position , preyPosition)
3   value = 10
4   while (dist > 1){
5     value = value / 2
6     dist = dist - 1
7   }
8 }

```

Figure 2.13: Pseudo code for a simple implementation of the predator/prey distance function. This implementation is based merely on the direct distance between the two.

I have, however, found it to give better results when diminishing the value based on the difference in horizontal and vertical direction (x,y), since the maze is a grid. I have further found it to give good results to diminish the value by $\frac{2}{3}$ for each integer difference in each direction, as shown in Table 2.1. The table should be read with the prey in the upper left corner, and each of the other fields corresponds to value relative to this position. E.g. if the prey were at position (0,4) and the predator were at position (2,3) the function would yield the result 4.44. Were the predator instead at position (1,0) it would yield 1.98.

The pseudo code for this approach is shown in Figure 2.14.

```

1 method (returns number):preyValue{
2   dist = Abs(position.X - preyPosition.Y)
3   value = 15
4   while (dist > 0){
5     value = value*(2/3)
6     dist = dist - 1
7   }
8   dist = Abs(position.Y - preyPosition.Y)
9   while (dist > 0){
10    value = value*(2/3)
11    dist = dist - 1
12  }
13  return value
14 }

```

Figure 2.14: Pseudo code for an improved implementation of the predator/prey distance function. The distance is here diminished by $\frac{2}{3}$ for each integer difference in the horizontal and vertical distance.

The size of the $Weight_{distance}$ constant for this function is of course relative to the weight constants of the other functions. E.g. it might be more important to move closer to the prey, than to maximize the number of available predator moves.

2.4.3.2 Prey function

The intention with the prey function is to introduce some means to identify limitations in the prey's move options. It is however not that relevant when only calculating a single predator move, and is thus not applied here, but it should definitely be applied in a game theory heuristic function, since it takes a set of positions into account and not just a single move.

2.4.3.3 Predator spread function

By increasing the distance between predators, it is ensured that predators spread out and cover more ground. A feasible way to implement this would be in the same manner as with the prey distance calculation, but for a one move calculation in a setup with no barriers it does suffice to ensure that the predators do not get in each others way. This is done by introducing a penalty value for positions which are reachable for more than one predator. The size of the value should be big enough for the predators to consider other positions, but not so big that none of the predators will move to the position, if it is favorable. A function for this could be defined as:

$$f_{predatorDist} = \begin{cases} W_{predatorDist} & \text{if } \text{Min}(2, \text{Dist}(\text{position}, \text{otherPredators})) = 2, \\ -W_{predatorDist} & \text{if } \text{Min}(2, \text{Dist}(\text{position}, \text{otherPredators})) < 2, \end{cases}$$

The *Min* function ensures that if the other predator is more than two fields away, it will not influence the function. By implementing it in this way, the function does not ensure spread, but it does ensure that the predators do not get in each others way, i.e. compete for the same position.

2.4.3.4 Field function

The advantageous position can be made as a predefined graph from the layout of the domain. In a maze with no internal barriers, the beneficial fields are all fields which are not along the boundaries of the maze, whereas in a real maze, beneficial fields will be fields which somehow cut off the paths of the prey while maximizing the options of the predator.

2.4.3.5 The heuristic function

By this approach the final heuristic function is:

$$Heuristic_{(x,y)} = f_{preyDist} + f_{predatorDist} + f_{Field}$$

The distance to the prey has been considered the most important function for this approach and its weight constant is therefore given the highest value. An example of a result of the heuristics on the fields in the maze is shown in Figure 2.15, where each field is assigned a value and thereby identifying the next predator move as the adjacent field with the highest value.

The pseudo code for a move is shown in Figure 2.16, whereas the complete pseudo code is included as Appendix A

This approach works very well in practice, and tests have been made with tweaked values for the above functions, which yields even better results. This strategy, however, led to consideration of a much simpler approach, which will be discussed in the next section.

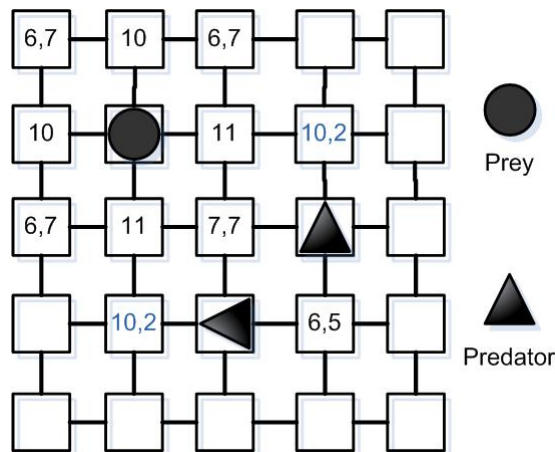


Figure 2.15: Example of the heuristics applied on the maze. The predators will move to the field with the highest value

```

1 Method (return void):PerformMove{
2   bestPosition = currentPosition
3   bestValue = fieldValue(currentPosition) + preyValue(currentPosition) +
4     predatorValue(currentPosition)
5
6   foreach (position adjacent to own position) {
7     if (position is inside maze){
8       positionValue = fieldValue(position) + preyValue(position) +
9         predatorValue(position)
10      if (positionValue > bestValue AND position not occupied){
11        bestPosition = position
12        bestValue = positionValue
13      }
14    }
15  }
16 }

```

Figure 2.16: Pseudo code for a single move my the heuristic evaluation

2.4.4 Winning strategy in a turn based game

The heuristic approach led to consideration of a winning strategy with only two predators, while assuming the approximation that the movements are turn based.

The strategy can be divided into the following three steps, given two predators and one prey:

1. **Step:** Regardless of the starting positions, the predators should place themselves in a position, where one predator is in parallel with the prey in the horizontal direction, and the other is in parallel with the prey in the vertical direction, as illustrated in Figure 2.17. It will always be possible to reach this situation, since the grid has fixed boundaries, and when the situation is reached, it will always be possible to maintain, since all robots move with the same speed.
2. **Step:** When step one has been reached, the predators move in on the prey until the prey is cornered at two sides (Figure 2.18). This situation will also always

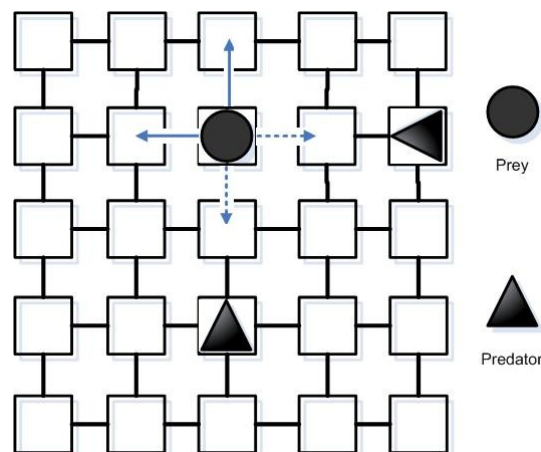


Figure 2.17: The predators are in parallel with the prey in both the south-east direction and the north-south direction

be possible to reach because of the fixed boundaries and because the robots move with the same speed. For each move the prey makes, the predators can make a counter move, which will either maintain the distance to the prey or diminish the distance to the prey. If the prey continues to move away from the predators, it will eventually reach a barrier and be forced to move closer to the predator. The prey will hereby eventually be cornered by the predators.

- 3. Step:** In the final step, the prey only has the option to move away from the predators, at which point the predators will follow (and satisfy (2)) and thereby continue to corner the prey at two sides. The prey will then first be cloistered by one of the maze barriers, and finally be trapped in a corner.

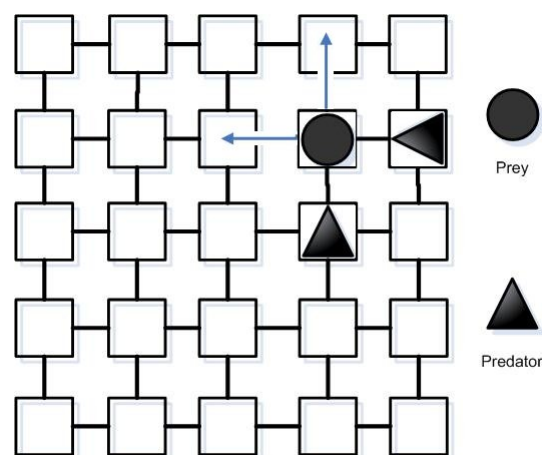


Figure 2.18: The prey is cornered on two sides

A segment of the pseudo code for this algorithm is shown in Figure 2.19 whereas the complete pseudo code is included as Appendix B.

This approach is, however, only guaranteed to work in a strictly turn based scenario with no internal walls. In practice, the prey will sometimes be able to make two

```

1 Method (return void):PerformMove{
2   // FIRST STEP: Align with the prey. Horisontal and vertical
3   if (not in line with prey AND other predator is aligned horisontally
4       with prey)
5       AlignVerticallyWithPrey ()
6   if (not in line with prey AND other predator is aligned vertically
7       with prey)
8       AlignHorisontallyWithPrey ()
9   if (not in line with prey AND other predator is not aligned with prey)
10      AlignWithPrey ()
11  if (horisontally in line with prey AND other predator horisontally in
12      line with prey OR
13      vertically in line with prey AND other predator vertically in
14      line with prey) // both predators in same alignment with
15      prey
16      AlignWithPrey ()
17  if (aligned with prey AND other predator is aligned with prey)
18      MoveToNextPositonCloserToPrey ()
19  // SECOND STEP:
20  if (in line with prey AND not adjacent to prey)
21      MoveToNextPositionCloserToPrey ()
22  // THIRD STEP:
23  if (in line with prey and adjacent to prey)
24      {} //Do nothing
25 }

```

Figure 2.19: A segment of the pseudo code for the winning algorithm in a turn based game. The segment shows the conditional statements for each move

moves, while the predators only make one, which puts the predators back to step 1. The prey will, however, not be able to make it past the predators, so the accessible part of the maze is diminished for each time, and the prey will eventually be trapped in a corner.

2.4.5 A mathematical simulator

While testing the different algorithms, it has proved useful to develop an additional simulator for testing the algorithms in a turn based single threaded environment. This is of course not enough to ensure that it works in the real environment, but it has been a good tool for testing that the theory of the approaches is implemented correctly. For this reason, the mathematical simulator is a single threaded simulator where each entity moves by turn. The simulator further allows to view the game in steps by taking one iteration (one move by each entity) at a time. The mathematical simulator is pictured in Figure 2.20.

2.4.6 Findings and results

The game theory approach would have yielded an optimal solution, but is unfortunately inapplicable to this setup, because of the unreliability of the robots and the huge number of calculations.

The heuristic approach does however yield a most beneficial solution, where the predators will narrow in on the prey and capture it in a small number of moves. This solution is however not optimal, but fairly efficient.

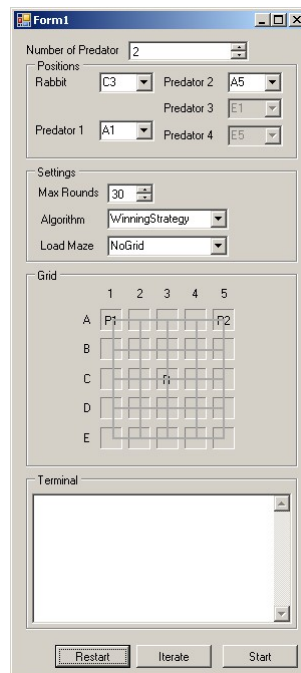


Figure 2.20: Picture of the mathematical simulator

The winning approach works extremely well, and will ensure capture of the prey in a minimum number of moves. Even though it assumes the movements to be turn based, it works well with the deviations in the setup. It also proved that the problem of capturing the prey can be accomplished with as few as two predators, and that this in an environment with no barriers is fairly easy.

These approaches have also proved that the problem is fairly easy to solve, and consideration on how to take the setup to the next level and complicate the capture of the prey is discussed in Chapter 5.

Chapter 3

Design and Implementation

Due to the limitations on the NXT the implementation is partly made in the MindStorms language and partly in *C#*. The part in the MindStorms language is limited to subroutines concerning the movement of the NXT, whereas the actual framework resides on a detached PC. Besides this, it is a requirement that the intelligence must be implemented as an isolated instance which can be swapped at run time. The implementation is currently made for only one PC, meaning that the agents do not run on their own machine, but are handled as separate threads on the same PC.

This chapter is in no way intended to be a thorough review of the design and implementation, but serves instead to give the reader a good insight into the program.

A brief overview

This chapter starts by outlining the design of the robot as a concept and defines the involved elements, followed by an overview of the *C#* components. Of these, I will go into detail on the agent part, since this is the most essential, and forms a good basis for the understanding of the implementation. Besides this, I will briefly illustrate the implementation of the algorithms and a game run.

3.1 The design of the concept

Because of the limitation on the NXT, the agent and its intelligent procedure need to run on an external PC. Besides this, the MAS requirements specify that the intelligent procedure must be allowed to be swapped at run time. These criteria make it appropriate to create a three part design, divided into a lower part, (the mechanical), a middle part (the agent) and an upper part (the AI). This design is illustrated in Figure 3.1.

To make the distinction of the different parts clear, a description of each is given below:

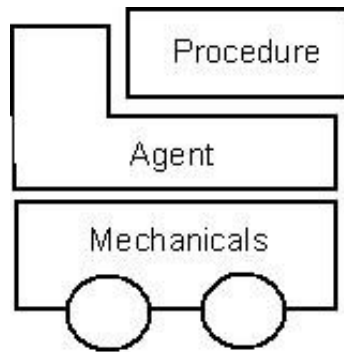


Figure 3.1: A graphical representation of the structure of the concept. The lowest part is the mechanical layer. The middle part is the agent and the upper part is the running procedure.

The **lower** layer consisting of the direct interaction with the NXT, i.e. performing actions as “drive”, “turn” etc. This layer should receive commands from the upper layer and relay responses back to this layer.

The **middle** layer is the agent entity, i.e. control layer. It monitors all the transactions between the upper and lower layers and maintains the current position of the entity. This layer is also responsible for relaying messages between the different entities in the network and negotiating for procedures. It is furthermore responsible for terminating the running procedure and evaluating on the objective and assign appropriate penalty values.

The **upper** layer, i.e. procedure is responsible for achieving the goal, and contains the algorithm for the approach. It also contains a local simulator for estimating the effectiveness of the algorithm, which is used by the middle layer when negotiating procedures. It is a requirement that the upper layer contains an initialization method and a run method, so that new procedures can be run by the existing entities.

The two lowest parts are a convention made necessary because of the nature of the NXT. It does however have the benefit that the mechanical part can easily be switched with another. It is the intention that the two lowest parts are closed parts strictly under the control of the manufactures, whereas the upper part is intended to be delivered by some third part developer, and can therefore not be trusted. All critical information, i.e. information related to the position, the number of moves, negotiation and trust values must not be editable by the upper layer. Furthermore, the upper layer should not be able to crash the lower part, and the middle layer should revert back to status quo if the upper part crashes. .

The benefit of having the procedure stored in this way, is that it makes the network highly dynamic, since new procedures can be introduced without taking down the network. Hence no down time is necessary.

3.2 The structure of the implementation

The implementation is divided into a number of different components as listed below. An abstract overview of the components are shown in Figure 3.2.

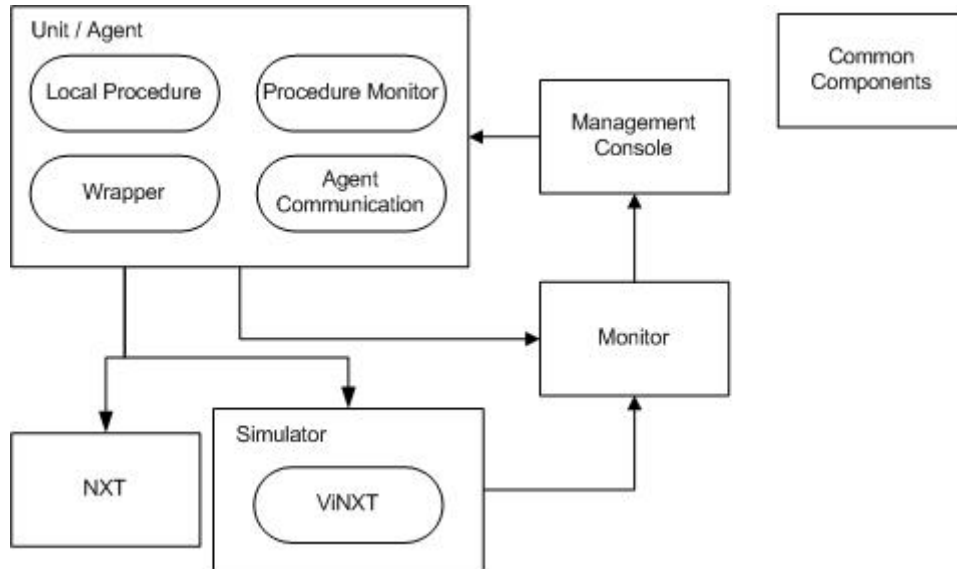


Figure 3.2: Abstract representation of the components and their relation. *Common Components* is a library which is used in all of the others.

CommonComponents is a library containing a number of useful tools developed during the process. This includes vector calculation, world representations, monitor references, NXT layouts.

Simulator contains the simulation of the domain. It is created as an isolated instance in order to avoid replicated errors, so that errors made in the other components will be caught by the simulator.

Management console which is a user interface for running the project. It contains controls for loading different maps, starting robots and monitoring the process.

Algorithms contains the different AIs for capturing the prey.

Unit refers to the actual agent. It contains wrappers for communicating with both the NXT and the simulator along with the simulated instance for communication between the agents. It further contains procedures on how to run the different algorithms.

3.3 The Agent

The agent is contained within the Unit component which consists of a variety of classes, which can be divided into the following subcategories:

Wrapper which interfaces with either the simulator or the NXTs. It is used by the agent to send commands to the robot and receive confirmations.

Agent management which contains classes for controlling the agent. It is responsible for communicating with other agents, for loading and terminating procedures and relaying information between the procedure and the robot.

Procedures which contains the different approaches for capturing the prey. The procedures are implemented as state machines containing distinct states for involved parts.

Communication which handles the communication between agents. This is handled by two separate classes: Master and Relay.

I will only go into detail on the *Agent Management* and the *Communication*, since the wrapper part is pretty straightforward and procedures are simply state machines, which either calls classes in the algorithm module or sends action commands to the robot.

The Agent Management contains a number of private classes besides its methods. These are **Entity**, **ProcedureEntity** **Master** **Relay** and **Unit**. The last three classes will be handled under *Communication*, whereas Entity is a simple wrapper object for storing information on the robot's position and orientation and ProcedureEntity class is a reference object used by the Procedures to get information about the world, the robot's position and communication.

The next section will walk the reader through the methods in the Agent Management.

3.3.1 Agent Management methods

The Agent Management is handled by the class **EntityAgent** and is allocated through an empty constructor. This is done due to thread issues, since I need to run all agent on a single computer. It does not have any functional reasons besides that. This approach requires that parameters are initialized through the **Initialize** method, takes the position and orientation of the robot along with information on which local algorithm to use as arguments. If run on physical robots, it also requires the port number. The whole process is started by the **Start** method.

The rest of the processes is internal and will follow the description below:

When started, the agent will load the local intelligence and then based on its own position, the other predator's positions and the prey, estimate how fast it can capture the prey. This value along with the strategy is then committed to a strategy pool, after which the agent waits for all strategies to be committed. When all strategies are committed the agents retrieve the selected strategy, and follow this for a limited period. This is based on an estimate made by the procedure, and an upper threshold defined by the agent. If the threshold is reached before the prey is captured, the agent will terminate the procedure and start over. This sequence is illustrated in the sequence diagram in Figure 3.3.

The loading of external procedures requires a bit more attention and is described in the following section.

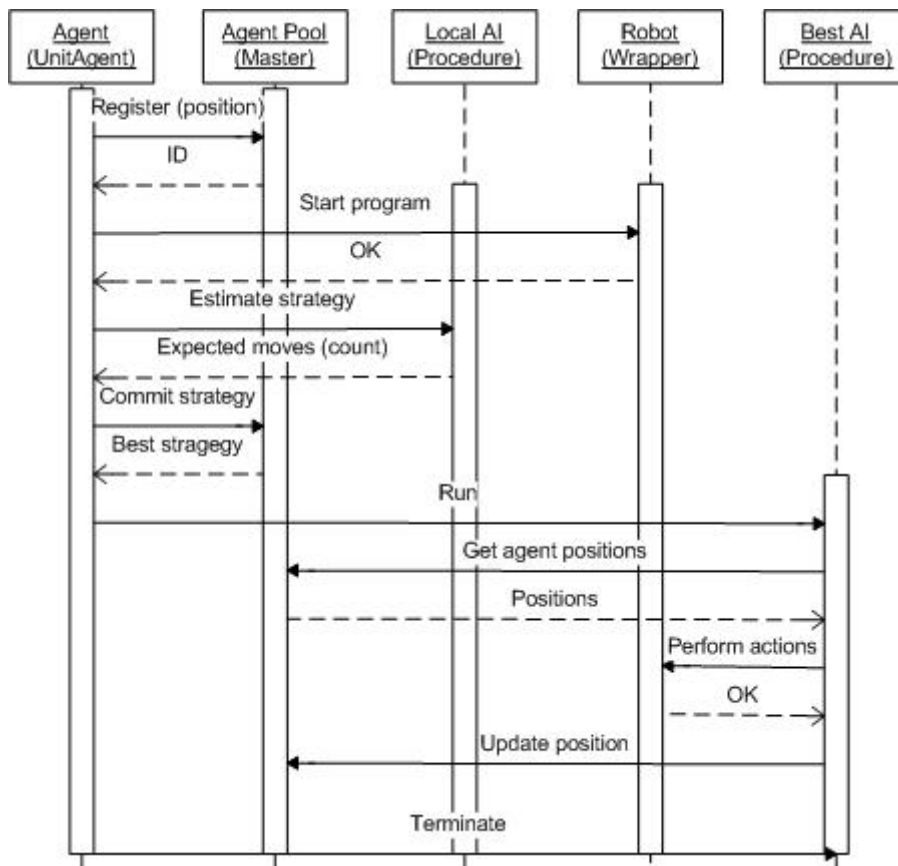


Figure 3.3: Sequence diagram of a game run. The section between “run” and “terminate” loops until the agent terminates the procedure or an end situation has been reached. The sequence diagram is a simplification of the actual process, since the procedures cannot communicate directly with other instances, but have to go through the agent. After a termination, the agent will evaluate on the process, and if an end situation has not been reached, it will go back to the local evaluation state and continue from there.

3.3.2 Loading of procedure

As stated in the design chapter each agent will represent its own procedure with the option of loading other procedures by the process of negotiation. To achieve this in practice, the procedures should be produced as libraries or DLL, and then loaded through reflection. The process is fairly simple and a sample code would look like the code in Figure 3.4

Initialise for initializing the instance. This method should take the parameters of the instance `World` and `Vector` which are both DLL made available in this project.

Run which should contain the run method, and run until the goal has been achieved or it is being interrupted.

```

1  try
2  {
3      // Instansiates the instance of the foreign intelligence
4      //Object foreignObject = Activator.CreateInstance(
5          foreignIntelligenceType);
6
7      //string id, Manager controller, Math.Vector position, Math.Vector
8      //orientation
9      Object foreignObject = Activator.CreateInstance(
10         foreignIntelligenceType, foreignIntelligenceArguments);
11     // Defines method references
12     //MethodInfo initialiseMethod = foreignIntelligenceReference.GetMethod
13     //("Initialise");
14     MethodInfo activateMethod = foreignIntelligenceType.GetMethod("Start")
15     ;
16     // Invokes the method
17     //activateMethod.Invoke(foreignObject, new object[] { });
18     activateMethod.Invoke(foreignObject, null);
19 }
20 catch (ThreadAbortException)
21 {
22     // do nothing
23 }
24 catch (Exception e)
25 {
26     throw e;
27 }

```

Figure 3.4: Sample code for loading dll through reflection

There is still the issues of distributing the DLL throughout the network, which could easily be done through any number of protocols and through serialization.

I have, however, chosen not to implement this, since the issues involved are purely technical and not really difficult. It is only time consuming and will require separate computers in order to perform any real practice tests. Since all agents currently run on the same computer, I have instead chosen to simulate this, by letting all the procedures reside in a common intelligence pool, from where the individual agents will be able to start the appropriate procedure after the negotiation.

3.3.3 Communication

As discussed in Section 2.3 a number of approaches exist for implementation communication in a MAS. The chosen one consist of a master tower, where each agent register at startup. This way the tower always has a complete list of all agents.

As another simplification, the tower is implemented as a local instance, where the agents communicate directly through C# methods. This is only possible because all the agent runs on the same computer. Ideally, the agents should have been distributed over a number of computers, and the communication should have been through the network protocol. It is still possible to perform the communication through the network, event though all agents reside on the same computer, but as discussed in Section 2.3, the implementation is straightforward but time consuming.

The negotiation between the agents are made by letting all agents calculate their estimate on capture of the prey and then submit their estimate to the Master along

with their trust table. The agent will then try to retrieve the best procedure from the Master, which will be locked until all agents have committed their procedure. When all procedures are committed, the first agent to make the request will then activate a calculation within Master, which will calculate the best procedure, and afterwards return that to all agents. The locks are shown below:

```

1 [MethodImpl(MethodImplOptions.Synchronized)]
2 public void CommitStrategy(string id, int threshold, Type objectRef,
3     Hashtable _trustvalues, string _strategy)
4     {
5         lock_strategyIsBeingCommitted = true;
6         // reset strategy
7         strategy = null;
8         strategies.Add(new ControlObject(objectRef, threshold, id, _strategy)
9             );
10        // update trust values
11        IDictionaryEnumerator e = _trustvalues.GetEnumerator();
12        while (e.MoveNext())
13        {
14            // Update the complete trust value list with the new values
15            if (trustValues.Contains(e.Key))
16                trustValues[e.Key] = (int)trustValues[e.Key] + (int)e.Value;
17            else
18                // Update the trust value list with values for the new agent
19                trustValues.Add(e.Key, e.Value);
20        }
21        if (strategies.Count == units.Count)
22            lock_strategyIsBeingCommitted = false;
23    }

```

```

1 [MethodImpl(MethodImplOptions.Synchronized)]
2 public ControlObject GetCurrentStrategy()
3     {
4         if (lock_strategyIsBeingCommitted)
5             return null;
6
7         lock_strategyIsBeingReceived = true;
8         if (strategy == null)
9             {
10            strategiesReceived = 0;
11            ControlObject bestObject = null;
12            int moves, bestMoves = Int32.MaxValue;
13            foreach (ControlObject co in strategies)
14            {
15                moves = co.ExpectedMoves;
16                if (trustValues.Contains(co.UnitID))
17                    moves += (int)trustValues[co.UnitID];
18
19                if (bestObject == null || moves < bestMoves)
20                {
21                    bestMoves = moves;
22                    bestObject = co;
23                }
24            }
25            moves = 0;
26            strategy = bestObject;
27
28            // reset trust values
29            trustValues = new Hashtable();
30        }
31        strategies = new List<ControlObject>(); // reset list of strategies
32
33        strategiesReceived++;
34        if (strategiesReceived == units.Count)
35            lock_strategyIsBeingReceived = false;

```

```

36|     return strategy;
37| }

```

The lock works by not allowing any agent to retrieve a strategy, while other agents have not yet committed theirs. When all agents have committed their strategy, the best strategy will be available to the agents until all agents have retrieved that strategy. It will not be possible to commit new strategies until all agents have retrieved the active strategy.

3.3.4 Implementation of the algorithms

The implementation of the algorithms is done through state machines, but any number of approaches can be used, since it is contained within a single object which only requirement is that it must have an initialize and run method with the stated parameters.

The state machine is divided into two different types of states. Normal state classes which are derived from an abstract state class. This class contains information on the successor class for a failure scenario and a success scenario. Besides these classes, a number of static classes exist for performing subroutines, i.e. turning and moving the robot.

A state diagram of a game run is illustrated on Figure 3.5. The procedure will terminate when the Endstate is reached.

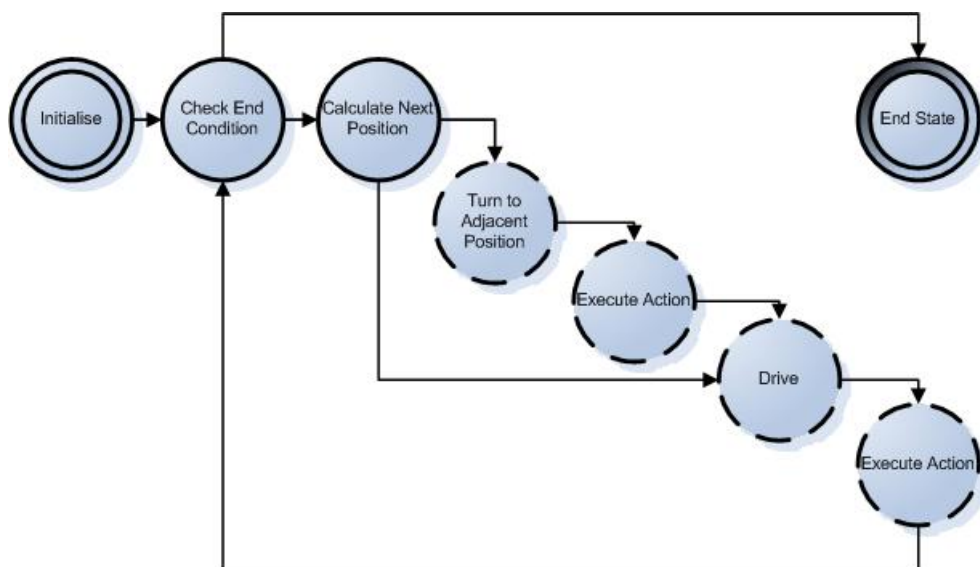


Figure 3.5: A state diagram for game run. The dashed states are static states containing sub routines.

3.4 Findings and results

The implementation is made with the proof of concept as the primary objective. For a ready to market product further effort should be put into isolating the different parts, and ensuring that changes can only be made in the intended way. Furthermore, efforts could be put into optimizing the system, e.g by performance testing as discussed in Chapter 4. But as a whole, the implementation proves that the concept works as intended.

Chapter 4

Testing

The testing of a project of these proportions is fairly extensive and time consuming, and since this is only a proof of concept and not a Ready For Market (RFM) product it has not been tested to its full extent. I have furthermore chosen only to perform tests within the scope of structural, functional and usability testing, where a lot of other test approaches exist¹. I will however in the following introduce the necessary steps for testing a product in depths.

4.1 Ways of testing

The most common and minimum requirements for testing are structural and functional testing, where in *structural tests*, also called white box testing, one assumes full knowledge of the program, and strives to test every line of the program code through test methods. The intention is to cover all possible outcomes and ensure correct output. It is further useful for testing against under- and overflow of data structures. A very common framework for structural testing is the Unit Test framework, which in Section 4.2 is applied to the Winning Algorithm as an example.

Functional testing is often called black box testing, where one assumes no knowledge of the underlying program and only tests that the program solves the problem it is supposed to, e.g. capturing the prey. This approach is applied in Section 4.3.

Then there is *usability testing* which concerns tests of the user interface, where one ensures that the necessary options are available to the user, and that these perform as expected. This approach is applied in Section 4.4. Usability tests can further be evolved to take the user experience into account as well. This has, however, not been applied here.

Further testing concerns for instance *code revision*² and *performance testing*³. I will

¹Peter Sestoft ([7] and [8] has written short and precise guides for testing through these approaches.

²Code revision, or code analysis, is tests where the code itself is reviewed for logical errors, unclosed loops etc.

³Performance testing, or profiling, is tests where advanced programs inject waypoints in the code and measure the time and resource consumption at run time.

not perform these two tests, since code revision should be performed by someone other than the programmer, and I do not have the tools for performance testing at my disposal.

4.2 Structural testing

As a very minimum when developing applications, structural tests should be performed on all classes and methods. The benefit of structural tests is to ensure expected behavior of all lines of code. It is particularly good at ensuring the right outcome of loops and conditional statements, since it is intended to probe all possible inputs. Structural testing further provides the great ability to pinpoint errors when modifying code, since a working test for old code can be applied on new code in order to ensure that it still operates correctly.

I have, however, not tested all my classes in the projects, since it is extremely time consuming and other aspects of the projects have been given higher priority. I have, however, included a structural test through the Unit Test framework of the `WinningPredator` class in Appendix D, and a small segment is shown in Figure 4.1

```

1  /// <summary>
2  ///A test for InLineWithPrey
3  ///</summary>
4  [TestMethod()]
5  [DeploymentItem("Algorithms.dll")]
6  public void InLineWithPreyTest()
7  {
8      bool expected;
9      bool actual;
10
11     for (int preyX = 0; preyX < world.HorizontalSize; preyX++)
12         for (int preyY = 0; preyY < world.VerticalSize; preyY++)
13             {
14                 for (int predatorX = 0; predatorX < world.HorizontalSize; predatorX
15                     ++
16                     for (int predatorY = 0; predatorY < world.VerticalSize; predatorY
17                         ++
18                         {
19                             // Re-initialise
20                             PrivateObject param0 = new PrivateObject(new WinnerPredator(
21                                 world, new Vector(predatorX, predatorY)));
22                             target = new WinnerPredator_Accessor(param0);
23
24                             if (predatorX == preyX || predatorY == preyY)
25                                 expected = true;
26                             else
27                                 expected = false;
28                             actual = target.InLineWithPrey(new Vector(preyX, preyY));
29                             Assert.AreEqual(expected, actual, "Prey_at_(" + preyX + ", " +
30                                 preyY + ") +
31                                 ",_Predator_at_(" + predatorX + ",_ " + predatorY + ")");
32                         }
33             }
34 }

```

Figure 4.1: Sample of the unit test class for `WinningPredator`

The code included in the figure tests the method `InLineWithPrey` which tests if the

predator is in line with the prey. It does this for all possible position combinations for both the predator and the prey, and performs the test for both horizontally and vertically aligned.

4.3 Functional testing

Functional testing is perhaps the most common and essential form of testing, where the tester tests that program actually does as it is intended. For this part I have performed test scenarios for most parts of the agent functionalities, which are listed below:

- Ensure that the prey will be captured with the heuristic approach
- Ensure that the prey will be captured with the winning strategy approach
- Ensure that the agent can switch between the different strategies
- Ensure that the strategies are assigned appropriate trust values
- Ensure that the program acts correctly both when running a simulation and in a real environment

It would be most preferable to have an automated test on this field, but I have not found the time to write such a test.

It is often helpful to list the test criterias in Test Data Sets, with reference number, input and expected output.

4.4 Usability testing

Usability testing applies to user interfaces, where the tester tests the outcome of changing the different settings in the interface. I have applied this approach to the graphical user interface (GUI), the `Management Console`, and ensured that it performs as expected.

The GUI is, however, pretty limited in its functionalities, since most actions are handled internally within the agents, and the GUI is primarily a monitoring tool. It is however ensured that the available functionalities work as expected. The GUI has been tested for the following:

- Defining local procedure on each agent.
- Defining starting position for each agent.
- Defining communication port (for the Nxt) on each agent
- Defining the number of predators (2-3)
- Control the prey in real time

- Define algorithms for a computer controlled prey
- Receive output from all agents
- Receive output from the simulator
- Switch between simulator and real environment

This part of the test should also ensure that the program reacts appropriately when the user misbehaves or makes an error, e.g. if a non existing port number or position is entered, or if two or more agents are initialized with the same starting position.

4.5 Findings and results

The project has been tested through structural, functional and usability tests. The tests are in no way intended to be exhaustive, so it is highly likely that errors exist which have not been discovered. The tests are, however, enough to prove that the concept works, and it has been ensured that the predators are capable of capturing the prey at all times, and further that the negotiation and switching of procedures work as intended.

It has also been ensured that the setup works in both the physical and virtual environment.

The tests have shown that the concept works as intended, but also that the real environment robots are very sensitive to irregularities in the environment. It would be very beneficial for the overall experience of the game, if these issues could be resolved.

Furthermore, if the implementation were to be extended to run on several computers, tests regarding the communication should be explored.

Chapter 5

Extensions to the initial setup

The first iteration proved especially two points: Firstly, that the physical setup of the robots lacked on three essential points, namely the ability to follow the path in the maze, the high sensitivity to roughness in the surface and the low speed. Secondly, that it was fairly easy to ensure capture of the prey with the current game rules, and a maze with no internal barriers.

The purpose of this section is thus to discuss different approaches on how to solve and evolve these two issues.

A brief overview

The first part of this chapter concerns improvements on the mechanical layer in order to overcome the robot's worst shortcomings. This is done through new physical parts in the form of caterpillar tracks and color sensitive light sensors.

The second part of this chapter concerns adjustments on the game part, where internal barriers are introduced in order to complicate capture of the prey, and adjustments are made on the algorithmic part to satisfy the modifications. Here included discussion on game theory reapplied on the new setup.

5.1 Extensions to the mechanical setup

The mentioned problems with the robot require a lot of time calibrating the sensors and ensuring an even surface in the maze, and even then, there is still a high probability that the robots will still get out off course for longer runs. This is very annoying and time consuming, but I would however not have treated this with high concern, if it was not for the introduction of new physical parts to the robot. I would otherwise have based the calculations on the simulator, but I find that it will be more interesting and viewer-friendly to perform the runs on the actual robots and have thus decided to explore this issue.

The new parts which have become available are caterpillar tracks and color sensitive light sensors. The caterpillar tracks ensure more accuracy on the turning of the robot,

and makes it more robust to unevennesses in the maze, i.e. the attached markings and the tape.

The current light sensors are simple sensors which measure the intensity in the reflected light, which makes the robot able to differentiate between, black, gray and silver colors. Since the path in the maze is currently colored gray, marked by black tape on either side, the robot is incapable of telling in which direction it is out of course when it strays from the path. When it gets out of course and encounters the black tape, it will start sweeping in both directions until it encounters the gray color again. This approach requires the robot to stop up and sweep every time it comes off course, and further limits the speed of the robot in order to minimize the risk of getting out of course. It also has the annoying limitation that it is very sensitive to changes in the surrounding light, and that the measurements differ from sensor to sensor, The sensors therefore need to be calibrated individually, and redone during the day as the daylight changes.

These issues can be summarized to the following requirements:

- Enable the robot to run accurately at higher speed
- Ensure that the sensors do not need to be calibrated
- Make the robot able to move between intersections without stopping and sweeping in order to get back on course

5.1.1 Discussion on the robot

The original tripod setup of the robot had the unfortunate property, that the rear wheel could turn in any 360° , which made its starting movement very uneven, which again required the robot to run with reduced speed. The introduction of caterpillar tracks very quickly proved that the speed of the robot could be increased significantly. A picture of new setup of the robot is shown in [Figure 5.1](#)

The original sensor did however still require the robot to stop up and sweep every time it encountered the black tape in order to get back on course. The introduction of a color light sensor was thus introduced in the hope to make the robot indifferent to the surrounding light, and eliminate the sweeping motion. The intention was that by differentiating the color on each side of the path, the robot would know if it was too far to the left side or the right side, and that it would hopefully enable the robot to continue moving while correcting the path.

5.1.2 The color sensors

The color sensors measure the relative light reflection in the colors blue, green and red. The first issues to be solved are thus which colors to use, and which colors make the best readings. I would need to use four different colors in order to indicate respectively:

- The left side of the path



Figure 5.1: The layout of the second iteration robot. The important difference is that the wheels has been switched with caterpillar tracks and that the light sensor have been switched with a color light sensor

- The right side of the path
- The path
- An intersection

I have through tests discovered that some colors will be overshadowed by others. This is due to the fact that the sensors perform relative measurements and this thus needs to be taken into consideration. The color sensors further provide the possibility to introduce gradients, which will make the robot able to calculate the angle at which it is off course, which would require some accurate form of calculation, e.g. a precise time interval at which the measurements are performed, which is not available on the NXT. The rotation degree on the wheel might, however, be used for this purpose instead. These issues have, however, not been pursued, since other methods have proved efficient for the current need.

5.1.3 The improved robot algorithm

With the introduction of new sensors and caterpillar tracks, it has been necessary to rewrite the line-following-algorithm, since it is now possible to determine at which side the robot is off course.

By letting the right side of the path be one solid color (i.e. either blue, red or green) and let the left side of the path be another color, the measurement can be made by

comparing the lack of one color, and thus determine if the robot is off course. Blue and red have proved to give the best readings.



Figure 5.2: Figure of a maze path where one side is blue and the other side is red

Pseudo code for the line following algorithm is shown on Figure 5.3 and the complete pseudo code for the NXT routine is included as Appendix C.

```

1 Module: FollowLine{
2     stop = false
3     while (stop = false)
4         red, green, blue = DO: MEASURE LIGHT
5         if (blue < threshold) // robot in red field
6             sweep = -1
7         if (red < threshold) // robot in blue field
8             sweep = 1
9         if ((red + blue + green) < blackThreshold) // robot at black
            point
10            stop = true
11        if (leftBlueRightRed == false) // value set in Main module
12            sweep = sweep * (-1)
13        switch (sweep)
14            -1:
15                DO: STEAR LEFT
16            0:
17                DO: DRIVE
18            1:
19                DO: STEAR RIGHT
20    }
```

Figure 5.3: Pseudo code for the line following routine on the NXT. The DO keyword indicates existing actions on the NXT

The algorithm works by measuring the intensity of light in blue and red respectively. If one of the colors is below a threshold, it means that the robot is too far off to one of the sides, and the robot can thus correct it by turning slightly to the other side. It also measures the total intensity of the three colors, which is used to determine if the robot has arrived at an intersection, which is marked by a black square. A picture of the new maze layout is shown in Figure 5.4.

The described approach does, however, only work if the blue color is always on the left side of the path and measures have to be taken in order to enable the robot to traverse the path in the opposite direction. I have found that the simplest way to do this, is just to let the robot perform a calibration at the beginning of each run, by first letting the robot turn 45° to the left, measure the color of the marking, and then calibrate from this. The nature of the maze is thus so that the color changes

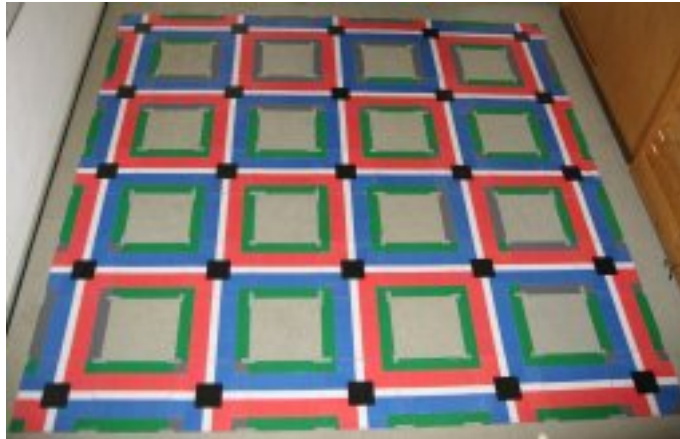


Figure 5.4: Picture of the mazed used with color light sensors

when the robot continues straight on from an intersection or turns completely around. The calibration is thus inverted each time the robot has turned any even number of directions (0 being an even number).

Since the robot knows at which side of the path it is, it does not need to sweep, and can thus continue to drive forward while getting back on course. This approach combined with the caterpillar tracks ensures that the robot is now able to satisfy the new requirements.

5.1.4 Findings and results

The introduction of caterpillar tracks and color sensors have proved highly efficient. It has both removed the need to calibrate the robots and the sensitivity to unevennesses in the surface, and thereby provided a reliable mechanical environment.

5.2 Extension to the game setup

The changes to the mechanical part highly improved the robot and made it capable of running smoothly. Further measures could be taken in order to make the robot run even faster, e.g. by attaching the wheel through gears. This is, however, not a primary concern but is left open for future extensions. I will instead here discuss possible extensions to the game setup.

The winning strategy for a turn based game proved that is it fairly easy for the predators to capture the prey in the current setup. The improvements to the mechanical part further ensures that the assumption of a turn based game is even more real. This has the side effect, viewed from a mathematical point of view, that it does not require much on the intelligence part. It would thus be nice to develop a setup, which would require more on the intelligence part. As an outline, it would be desirable with a setup that as minimum would include search functions, planning and estimations, whereas the current implementation uses direct distances, only calculates one move ahead and operates with absolute success criterias (capture vs. no capture).

5.2.1 Defining the problem

The primary issues at hand are, that given the current set of game rules and the current environment (the maze) it is too easy for the predators to capture the prey. Measures must therefore be taken which can complicate the process of capturing the prey. This could either be done by changing the game rules or by changing the environment.

Changes to the game rules could for instance be introduction of more than one prey or assigning win conditions to the prey. The problem is, however, that by introducing two preys, the predators could simply just focus on one prey, and afterwards just capture the other. This would not require any additional intelligence, whereas a win condition for the prey is difficult to define within the boundaries of the predator/prey domain. It could be an introduction of critical fields, which would add minor extra dimensions to the intelligence, since predators now need to protect areas as well as capture the prey. I do, however, find that changes to the environment are more beneficial to the project and provide a better overall experience for the viewer. I will therefore in the following discuss concerns on how to adapt the system to an environment with internal barriers, i.e. an actual maze. Further thoughts on changes to the game rules are, however, discussed in Chapter 7.

5.2.2 Heuristics - Reapplied

When introducing internal barriers to the maze, the current heuristics are rendered useless. Primarily because the predator/prey distance function is based on the direct distance between the two, which might now be blocked by barriers. It is, however, a simple change to adapt the function to some shortest path algorithm instead. By still maintaining that the heuristic shall calculate values for the fields adjacent to the predator position, an algorithm which considers this is required. In normal cases, where one wants to find the shortest path between two points, the A* algorithm is usually considered the best. It does, however, not fit directly for my needs, unless I call the algorithm for each position adjacent to the predator, which will give repeated calculations. Adjustment could however be made to overcome this, but I will instead consider Breath-First-Search for my needs, since this algorithm works without adaptations¹.

The changes to the predator/prey function are sufficient to make the predators narrow in on the prey, but it will not ensure capture, since maze layouts exist where the predators might just tail after the prey without ever catching it. Modifications therefore also need to be made to the field function. I have here considered a setup where positions which somehow cut off parts of the maze are considered favorable, as illustrated in Figure 5.5. These positions are identified by running both a horizontal and vertical analysis of the maze, where positions which block a narrow path are identified.

I have through tests observed that predators with the heuristics modified in this way are capable of capturing the prey in most scenarios. Further measures should,

¹A number of search algorithms are discussed in Introduction to Algorithms [3] and [6].

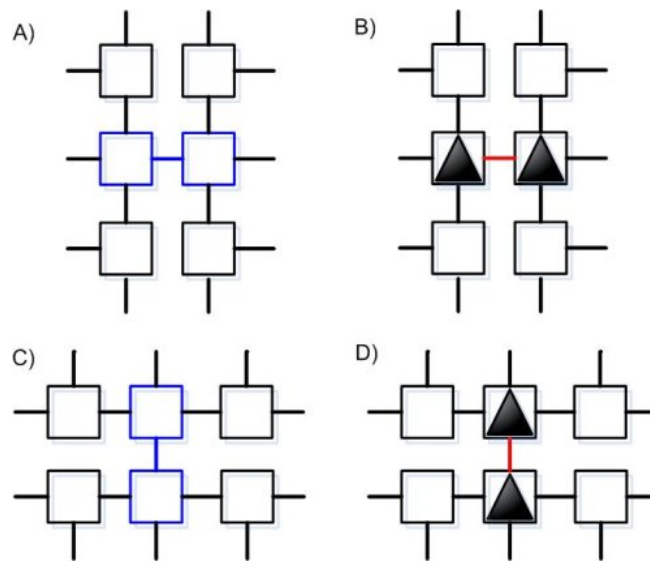


Figure 5.5: Figure A and C show passages which can be blocked by positioning predators in either of the two positions shown in figure B and D. These positions are therefore given a higher value by the maze function

however, be applied to ensure winnings at all times. A highlight of the results in this matter is found in Chapter 6.

5.2.3 Game theory - Reapplied

The game theory approach was originally discarded due to the problems on the mechanical part and calculation restraints. I do, however, think that the improvements on the mechanical part along with minor adjustments on the program and the fact that only two predators are needed, have opened the possibility that the approach might be applicable after all, which I will describe in the following.

By reducing the number of predators from four to two, the branching factor has been reduced from 5^4 to 5^2 . The game tree does, however, still grow big very fast, but measures exist to deal with this.

First of all, the game tree can be reduced by removing permutations, i.e. identical position combinations, except that the predators have switched place. Since it does not matter which predator is on either of the two positions, there is no need to distinguish between the two permutations (see Figure 5.6 for example). It is therefore possible to reduce the size of the tree, by checking for these permutations, and only add the position as a node in the tree, if it does not already exist as a permutation.

Secondly, situations will occur where the exact same positions can be reached at different depths of the tree, and thereby resulting in identical subtrees within the tree. There is no need to duplicate these, which can be avoided by transforming the game tree into a game graph, where reoccurring positions simply refer back to an existing node. By introducing these measures, the game graph is limited to the total number of possible position combinations, which in a five times five grid is

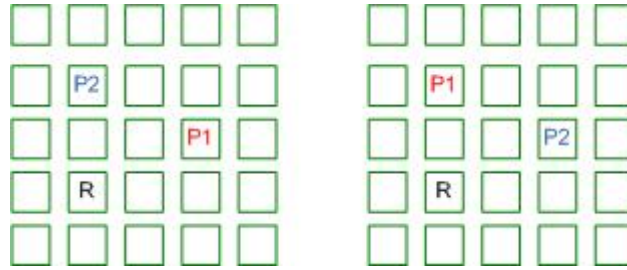


Figure 5.6: Example of a position permutation, where the same three positions are occupied, but where the predators have switched place

$$25 \cdot 24 \cdot \frac{23}{2} = 6,900 \text{ combinations.}$$

But let us consider how to build the tree in the first place. We will still be restrained by computational power to some extent, so it is not possible to build the whole tree, at least not in one go, and we therefore need to cut off the tree prematurely, and call an evaluation function. The previous discussed heuristic will be suitable here. It should be possible to build the tree to some depth, and then search the available tree for the next move. Additional computational power could also be freed by changing the implementation, since each agent is currently a single thread, which calculates the next move based on the current positions, after which it instructs the mechanical part to execute the move. It here waits for the robot to move from one position to another before it performs any more calculations. This process of moving takes around three seconds, in which the procedure is just idle. By letting the agent continue its calculations in this interval, it will most probably be able to build the tree to a reasonable depth.

Assuming that the tree has now been built to some extent, the agent still needs to search through the tree in order to determine the next move. When the tree becomes huge, this will also become a costly procedure, if all nodes need to be searched, and it is therefore necessary to reduce this. A game tree search is normally performed as a minimax search, which evaluates the lowest nodes in the tree and propagates the result back up. This search can be effectively reduced by using alpha-beta pruning, which is used to estimate if a branch is feasible or not². As an example, consider Figure 5.7.

The figure illustrates a game run for a single turn, where the predators move first. The first depth contains all 25 possible predator moves, though only two are shown. The next level contains additional five prey moves for each of the above predator moves, where only three are shown in the figure. At this depth the tree build is terminated. Alpha-beta pruning is then called on the tree which traverses down the tree until it reached a terminal node (leaf), at which point it will call the heuristic evaluation function to get an estimate of the advantage of the position. When it then traverses the other branches it will then compare the values and then abort the search down that branch if the value is unfeasible. For the example in Figure 5.7, the algorithm traverses down the left branch and evaluates the leaves. The best value is

²Benefits and drawbacks on minimax search and alpha-beta pruning are discussed in [6] and a tutorial on implementation is found at AI-Depot [2].

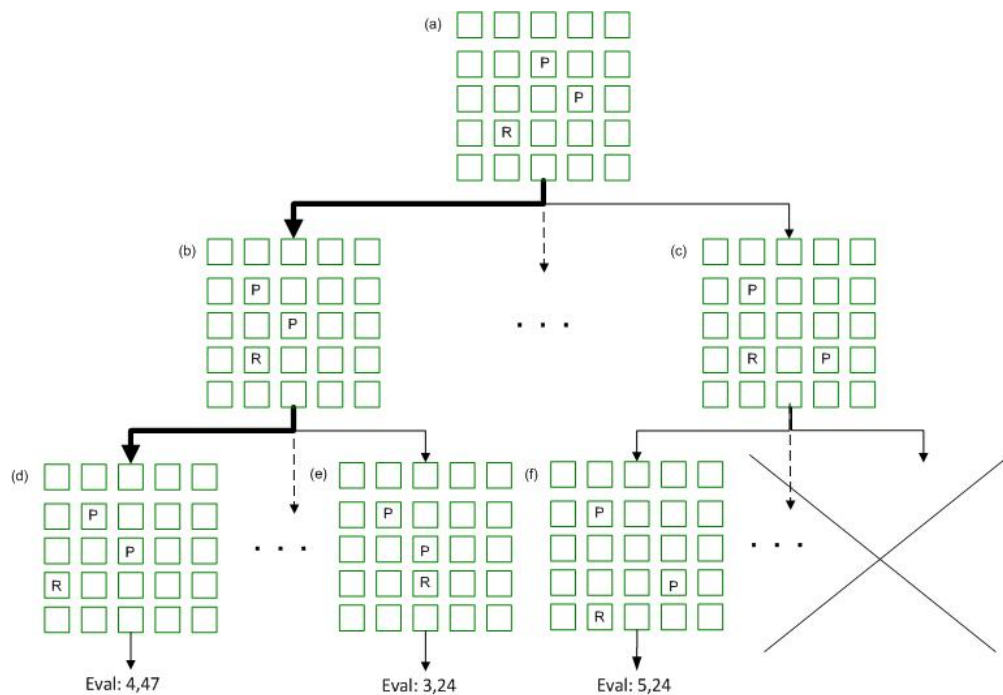


Figure 5.7: Example of cut off game tree pruning. The tree will be built to a depth of two, and use the heuristic to evaluate the position. In this case, based on the overall distance to the prey from the predators.

here 4.47, since it is a prey move and the prey wants to maximize the distance to the predators, and the value is then store for later reference. When the algorithm afterwards traverses down the right branch it evaluates a position to 5.24, and since this value is bigger than 4.47, the prey will choose this one if it gets the option, and it is thus not feasible for the predators to traverse down this branch and there is therefore no need to evaluate that branch any further. By this approach, the predators can identify the best move to be along the left branch, since this will yield the most feasible position.

The effectiveness of the pruning method is very depending on the order in which the branches are examined.

By enforcing these measures, I find that it should be possible to apply game theory on the domain, though I have not had the time to do so.

5.3 Findings and results

The changes to the mechanical setup have been fully implemented, and the other parts of the project all run on the improved setup, and have been tested to run satisfactorily.

The introduction of barriers is implemented to some degree. The framework now supports it, and the predators are capable of capturing the prey in most cases. They do that by following the adapted heuristic approach, which in its current state is

not sufficient to ensure capture of for all maze layouts. The setup has been tested on the maze layouts shown in Figure 5.8, where the mix has proved to be the most challenging.

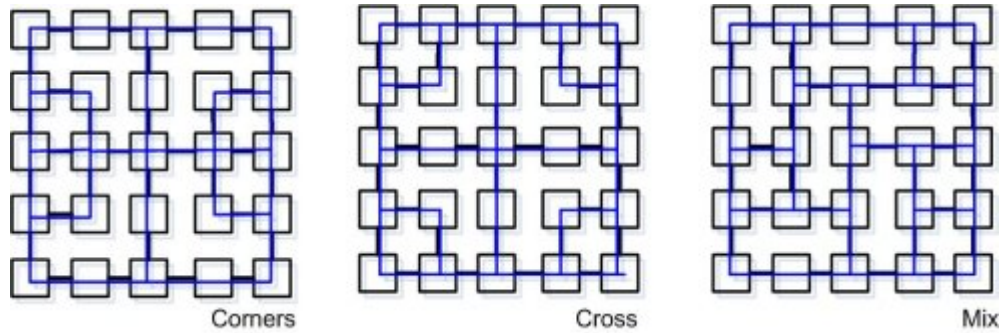


Figure 5.8: New maze layouts

The next step in this direction should be to actually implement the game theory approach and thereby ensuring capture of the prey for all layouts.

Chapter 6

Results

This chapter serves to highlight the results and discoveries made during the project, especially regarding the algorithmic implementations in the different environments, where both a non barrier layout and different barrier layouts are considered. The content in this chapter is backed up by the videos included on the CD.

In all the discussed approaches, the predators are capable of capturing the prey in all the tested environment layouts. Some of the approaches do, however, suffer from minor shortcomings, as will become clear in the following.

6.1 The heuristics

The idea behind the heuristics, was first as an heuristic for the game theory to evaluate the advantage of a position, and later as an isolated procedure for estimating one move ahead. The “SimpleHeuristic” approach is as the name implies a naive implementation of the heuristic functions, and will in most situations ensure that the prey is captured. It does however suffer from deadlocks, since the implementation is symmetric, so predators might choose just to follow the prey in parallel as illustrated in Figure 6.1. This has however been solved in the “TweakedHeuristic” implementation, by assigning appropriate values to the field function, but can also be solved by making the prey function asymmetric. One should, however, keep in mind, that the heuristic procedure was originally developed for at setup with four predators, and the situation where two predators follow in parallel with the prey is perfectly acceptable, as long as a third or fourth predator will come along and block the prey’s path in one of the other directions.

As videos show, the heuristic approach is perfectly capable of capturing the prey in an no barrier environment with as few as two predators.

When applying barriers, the heuristics still work, but they now suffer from their lack of planning, since the predators now have no means of estimating the consequences of blocking each others path. The heuristic can though still be modified to suit a specific layout, but it will no longer yield optimal solutions when applied to other layouts. Consider for instance the earlier problem from Figure 6.1 (case 1) but now applied

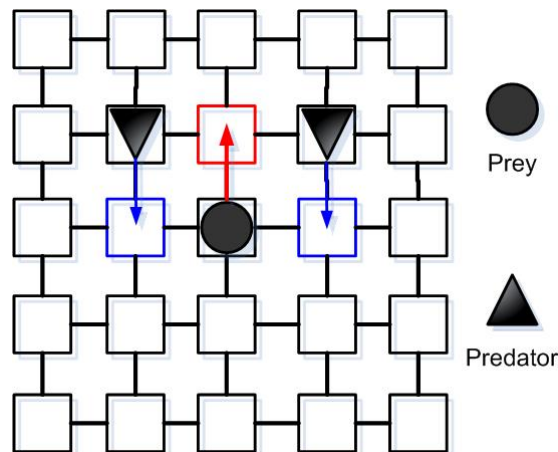


Figure 6.1: Deadlock scenario for the “SimpleHeuristic” procedure. Both predators will choose the blue field, whereas the red field would be the optimal. The “TweakedHeuristic” procedure does not suffer from this problem.

to a cross layout as illustrated in Figure 6.2 (case 2). In the first case, the field value would ensure, that one of the predators would move to the red field of Figure 6.1. In the second case, however, it is not indifferent which predator moves where, since it in the wrong case will yield a deadlock.

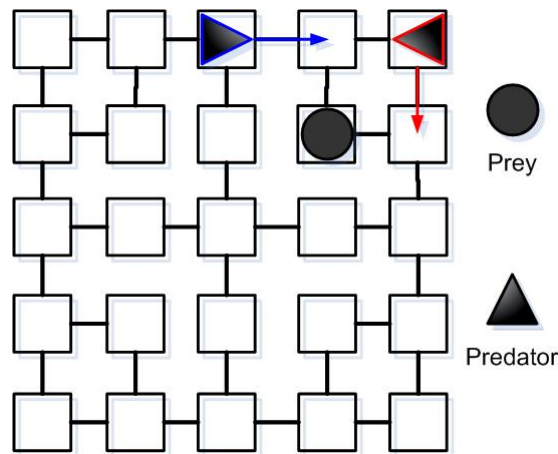


Figure 6.2: Example of the same problems as illustrated in Figure 6.1, but here it is essential which predator moves to which position, since it might otherwise lead to deadlock. The predators should move as the arrows indicate.

Another example to consider is the one shown in Figure 6.3. Here both predators compete for the same field, but it is only optimal if the predators move as the arrows indicate.

The “EdgeHeuristic” procedure has been tweaked to handled these specific cross problems and will yield an optimal solution of this layout. It will also perform well when applied to other layouts, but not always optimal. My point is, that though it is possible to modify the heuristics to suit a specific layout, it is not possible to ensure optimal solutions for all layout through one move heuristics alone. For this purpose

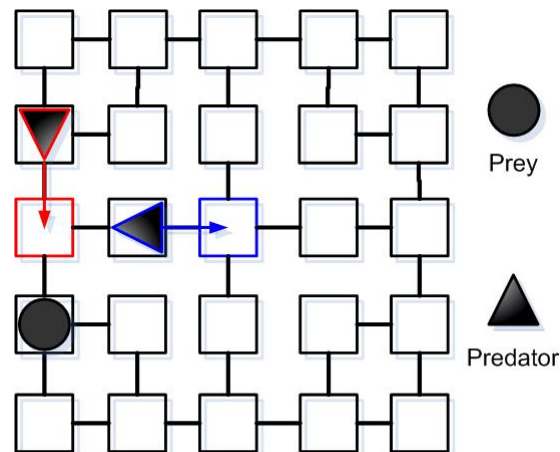


Figure 6.3: Example of a cross specific problem, where it is important which predator move where in order to ensure an optimal solution. The predators should move as the arrows indicate.

an element of planning is required. As an example consider the layout in Figure 6.4. This layout practically only exist of t-crosses, and the predators are here required to know the consequences of blocking each others way for nearly every intersection.

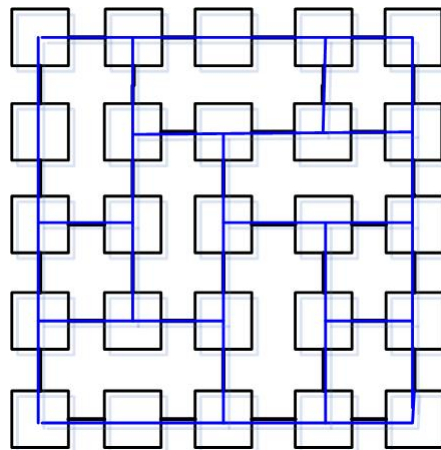


Figure 6.4: Example layout where an element of planning is required in order to ensure optimal solutions.

The “EdgeHeuristic” applied on this layout might with only two predators result in deadlocks, but will with three predators be able to capture the prey in an efficient manner. Some example runs on this scenario is included on the CD.

6.2 The winning strategy

The winning strategy works perfectly well in a environment with no barriers, where it will capture the prey in an optimal way, as the videos on the CD show. The

winning strategy is only illustrated with two predators since no more is needed. It does however not work as soon as barriers are included, but this is only as expected.

6.3 Agent negotiation and procedure swapping

The agent negotiation and procedure swapping works as it is supposed to. The agents will negotiate for the best procedure, and that procedure will be duplicated on to all the agents.

The method for evaluation the effectiveness of a procedure is, however, not that feasible. Since the agents currently do not have any means of planning, their only means of estimating the effectiveness of a procedure is by running it in a simulated environment. For this purpose, they only have a random moving prey at their disposal, and since a random moving prey does not perform optimal moves, the end estimate is not accurate. In an environment with no barriers, it does not matter much, but when applying barriers, the estimates are very inaccurate.

6.4 The robot

The robot has been improved, so it no longer suffers from previous projects shortcomings. It will now take a constant time to drive from one intersection to another, and is not vulnerable to unevennesses in the surface. The improved sensor also ensures that the robot will not drive of out of the maze.

Chapter 7

Further developments

The predators can in the current implementation capture the prey in a maze with no barriers in an optimal way, and can to some extent capture the prey in a maze with internal barriers, dependent on the layout of the maze. I am, however, certain that an implementation which includes some means of planning, for instance game theory, will yield an efficient solution for all layouts. Other aspects therefore need to be considered for further developments.

For the MAS part, the most obvious extension would be to implement the agents on distinct computers, and thereby letting each agent reside on its own computer. For this part it will also be interesting to scale the project to a much larger scale and thereby letting the communication between the agents play a much more important role. The MAS could also be extended with a security analysis, and include malicious agents, and make the system able to handle these.

As for the game part, further measures need to be taken in order to complicate the problem of capturing the prey, since the discussed strategies suffice for the current setup. As I see it, measures could be taken either by changing the environment or by changing the game rule or both. As for the environment, various kinds of barriers could be considered. For instance barriers, which can only be traversed in one direction, or dynamic barriers in the form of doors which can be either opened or closed. The trigger mechanism could then either be time related or as positions within the maze, which again opens the possibility for allowing the prey to trap the predators, and thereby introducing a win scenario for the prey. I do, however, find that changes to the game rules will be the most relevant element to explore for further developments as discussed in the following section.

7.1 Changing the game rules

I think that further developments on this project should yield higher requirements on the involved strategies. I do, however, find it difficult to define a suitable change to the game rules, which would increase the requirement on the intelligence and still fall within the definition of a predator/prey domain. I will, however, in the following

go a bit out of the way and discuss changes which fall within the boundaries of the elements in the project, but not exactly the definition of the domain.

The domain limitation is a limitation set by the title of this thesis. But if we however consider scenarios outside of the domain, I find that a more even-game-like scenario where both sides have the same advantages and disadvantages would be interesting, i.e. competing as teams. I imagine that the game should be extended to include objects or goals, which the two teams should compete over. For this part, a lot of existing computer games could be used as inspiration. I have especially two scenarios in mind:

Firstly, that a number of fields (say three) are fields of interest, where the robots need to visit these fields in order to reserve them. And the field is reserved by the team has have last visited that field, and the winning team will be the first team to have visited all fields last. In computer games this scenario is often referred to as “Domination”.

The setup is interesting because it challenges the intelligence on elements as estimation, should I stay close to one field of interest or should I move toward another and let this field stay open for the other team? Strategies for best to defend a field also apply, e.g. how to block the path of the other team.

As an addition, rules regarding capture of the other team could also be applied. It could for instance be as in the current setup, or it could be in a setup more like draughts, where a robot (piece) is captured if the other team has a robot on two sides of the captured robot. The win scenario could then either be to capture one of the other team’s robots or occupy all fields of interest.

Another set of game rules would be some kind of collection game, where objects are placed at random throughout the domain, and the two teams should try to collect the most objects. This approach would include a lot of planning and shortest path strategies. The objects could here again either be fields of interest or actual objects which should be moved to a position outside of the maze. Strategies could again either be just to collect as many objects, or let one team robot collect objects while the other blocks the path for the other team.

A number of different strategies apply to the above mentioned, and my point is that by changing the game rules to be more like an actual even sided game, it will open a lot of extra possibilities within the field of artificial intelligence, which, as I see it, is not possible in a normal predator/prey domain.

Chapter 8

Conclusion and perspectives

This thesis has so far discussed subjects on the matter of robots, multi agent systems, artificial intelligence, procedures, simulation, design, framework and tests. I have, in the process of making this project, undergone a lot of different processes. Here included narrowing down and identifying essential aspects. Construction and automation of robots. Considerations on feasible multi agent system implementations, such as strategies, communication, negotiation, execution and evaluation. Considerations on design and requirements on developing suitable simulators for the defined domain. Identification, implementation and testing of AI strategies. And finally inter-component-communication, making it all work together, and testing that it all behaves as expected.

Though measures can still be made to improve the project and make it capable of handling more complex problems, the current implementation solves the defined problem in a most efficient way.

I will now summarize it all in a final conclusion along with some perspectives.

The project has dealt with three primary elements of concern. First of all, **the robots**, which are now capable of navigating the environment in a most convincing and reliable manner, have, from their initial construction to the end setup, been improved significantly. They do no longer drive off the path, and although they might stray a bit, they will always arrive at the intersection. The improvements also ensure that the travel time from one intersection to another is standardized, and performed much faster than in earlier solutions. Furthermore the robots no longer need to be calibrated before each run, and are no longer sensitive to changes in the surrounding light. In connection with the robot's improvements, the physical environment has been modified to correspond with the changes. The path is thus now identified by distinct colored sideways.

I will also point out that the mechanical setup is much more superior to previous setups developed in this lab, but of course, the previous groups did not have the same mechanical parts at their disposal.

The second element of concern was the implementation of a framework which would allow dynamical adaptation of new procedures. This has been implemented as a **multi agent system**, where each agent, after the negotiation process, possess full knowledge of the procedure, and thereby eliminating any master/slave relation. This approach ensures maximum stability although it comes at the cost of resources, but it is in this way ensured that the procedure will be carried out regardless of the representing agent's own stability.

The multi agent system is further implemented in such a way that agents, when submitted to the system, register themselves in a common pool, and thereby making their own local procedure available to the rest of the system through the negotiation process. The system is thus automatically updated with new procedures simply by submitting new agents.

The framework also serves to prove the proof on concept, that this implementation is intended as, where procedures are represented by agent entities, and are equally negotiable within the domain. The procedures will thus be loaded based on their estimated effectiveness, and will afterwards be evaluated and trusted in regard to their achievements. The process of estimating, negotiating and evaluating would need to be modified to suit any other need than that of the current implementation, but the framework is otherwise applicable.

The third element of concern is the **artificial intelligences**, which are not exactly chess computers, but they are still capable of challenging and excel human players and thus ensuring capture of a human controlled prey. But of course, the predators also have the advantage in this domain.

The implemented AIs turned out to be relative simple, but highly effective, procedures. As it turned out, a heuristic, which only plans one move ahead for a single predator, and only considers positions immediately adjacent to that predator, is sufficient for capturing the prey in various maze layouts. It performs well with as few as two predators, but for some maze layouts more predators are needed. If only two predators are permitted, an element of planning is though required. As for this, game theory is considered feasible, and an implementation approach has been outlined.

As a final note on this element, it was discovered during the process, that the solution in a no-barrier-layout is pretty straightforward, and the solution for this is considered a winning strategy within these parameters.

These three elements of concern have been implemented in a **three tier design**, which has the important property, that it is easy to switch any of the layers with other components. The framework resides as the middle layer, whereas the AIs are isolated in the upper layer and can easily be replaced with other problem solving approaches. The evaluation mechanism in the middle layer though needs to be updated if other success criteria are introduced. The mechanicals are in the same manner isolated to the lowest layer, and the NXT can thus easily be replaced with any other kind of robot. For instance the SMRs¹ from Department of Electrical Engineering.

¹The SMRs are basically motherboards on wheels. They consist of high tech components, but require a lot of effort on automation in order to perform well. They are also rather big and not particularly customizable.

The whole concept is proved to run in both a physical environment with actual robots in the form of Lego MindStorms NXTs, and in a virtual, **simulated**, environment. This serves to prove the concept in more than one domain, and further ensures independence between the layers in the three tier design. It also has the great advantage that testing can be performed in a faster and more reliable way, than if run in the physical environment. As for this purpose, both a purely mathematical single threaded simulator has been developed for testing the IAs in an isolated environment, as well as a multi threaded simulator has been developed for testing the whole domain.

As a final note on the whole development, the concept has been **tested** to perform as expected. The test has, however, been limited to a minimum, but concerns on approaches for thorough testing are included.

Perspectives

It should be possible to implement the heuristic navigation approach in any domain with the purpose of navigating in the environment without getting in each other's way, because, though capture is not ensured to be optimal, the agents will still navigate in an efficient way.

By this approach, one practical implementation could for instance be in factories, where robots need to navigate in the same environment without hindering each other. The benefit from this approach is that only a small number of calculations are needed, and that this method thus can be implemented on lesser hardware, in opposition to complicated behavior patterns which require much more computational power. Another implementation could be on automatic cars, as seen in some American amusement parks. The heuristic approach can be implemented as a simple routine on robots performing any number of tasks, while working in the same environment. The routine will simply ensure that they do not get in each other's way.

As for the framework, this can be implemented in any domain where a number of agents cooperate on objective goals, and might for instance be applicable in "search and rescues", where the agents represent different strategies which are particularly applicable in certain environments, i.e. forest terrain, open landscapes, water etc.

This concludes my thesis on **Artificial Intelligence in the Predator/Prey Domain**.

Bibliography

- [1] Christian Agerbeck, *Multiagent Robot systemer*, Tech. report, Technical University of Denmark, spring 2007, Special courses at IMM.
- [2] AI-Depot, *Minimax explained*, <http://ai-depot.com/articles/minimax-explained/1/>, 2007.
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, second edition ed., The MIT Press, 2001.
- [4] James F. Kurose and Keith W. Ross, *Computer networking*, Addison Wesley Longman, Inc, 2001.
- [5] Anders Lemke, Johan Laidlaw, and Lars Zilmer-Pedersen, *Developing Multi-Agent Lego Robotics*, Tech. report, Technical University of Denmark, spring 2007, Polytechnical Midterm Project at IMM.
- [6] Stuart Russel and Peter Norvig, *Artificial Intelligence - A Modern Approach*, Prentice Hall, 2003.
- [7] Peter Sestoft, *Systematic software test*, Tech. report, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Denmark, 1998.
- [8] Peter Sestoft., *Systematic software testing*, Tech. report, IT University of Copenhagen, Denmark, 2008.
- [9] Seti@Home, <http://setiathome.berkeley.edu>.
- [10] Robin Sharp, *Principles of protocol design*, draft 2. ed., IMM-Tryk, DTU.2005, 2004.
- [11] Michael Wooldridge, *An introduction to multiagent systems*, John Wiley & Sons, LTD, 2002.

Appendix A

Pseudo code for the Heuristic

```
1 // Pseudo Algorithm for the heuristic strategy
2
3 Method:HeuristicStrategy(allPredatorPositions, preyPosition){
4     ownPosition = getOwnPosition(allPredatorPositions)
5
6     while (PreyCanMove)
7         PerformMove()
8 }
9
10 Method (return void):PerformMove{
11     bestPosition = currentPosition
12     bestValue = fieldValue(currentPosition) + preyValue(currentPosition) +
13         predatorValue(currentPosition)
14     foreach (position adjacent to own position) {
15         if (position is inside maze){
16             positionValue = fieldValue(position) + preyValue(position) +
17                 predatorValue(position)
18             if (positionValue > bestValue AND position not occupied){
19                 bestPosition = position
20                 bestValue = positionValue
21             }
22         }
23     }
24
25 Method (returns number):fieldValue{
26     if position is not along maze boundaries
27         return 1
28     else
29         return 0
30 }
31
32 Method (returns number):preyValue{
33     dist = distance(position, preyPosition)
34     value = 10
35     while (dist > 1){
36         value = value / 2
37         dist = dist - 1
38     }
39 }
40
41 Method (returns number):predatorValue{
42     if position is reachable for other predator in one move
43         return 0
```

```
44     else
45         return 2.5
46     }
47 }
48
49 // IMPROVED PREY VALUES FUNCTION
50 method (returns number):preyValue{
51     dist = Abs(position.X - preyPosition.Y)
52     value = 15
53
54     while (dist > 0){
55         value = value*(2/3)
56         dist = dist - 1
57     }
58
59     dist = Abs(position.Y - preyPosition.Y)
60
61     while (dist > 0){
62         value = value*(2/3)
63         dist = dist - 1
64     }
65
66     return value
67 }
```

Appendix B

Pseudo code for the Winning Algorithm in a Turn based game

```
1 // Pseudo Algorithm for the Winning strategy in a turn based game
2 // The code assumes that the layout of the world is a grid which is viewed
3 // as a normal mathematical coordinate system. It is further asumed that the
4 // boundaries of the world is known
5
6 Method:WinningStrategy (ownPosition , otherPreatorPosition , preyPosition) {
7     while (PreyCanMove)
8         PerformMove()
9 }
10
11 Method (returns boolean):PreyCanMove {
12     if (position left of prey is free AND is inside world OR
13         position right of prey is free AND is inside world OR
14         position above prey is free AND is inside world OR
15         position below prey is free AND is inside world)
16         return true
17     else
18         return false
19 }
20
21 Method (return void):PerformMove{
22     // FIRST STEP: Align with the prey. Horisontal and vertical
23     if (not in line with prey AND other predator is aligned horisontally
24         with prey)
25         AlignVerticallyWithPrey()
26     if (not in line with prey AND other predator is aligned vertically
27         with prey)
28         AlignHorisontallyWithPrey()
29     if (not in line with prey AND other predator is not aligned with prey)
30         AlignWithPrey()
31     if (horisontally in line with prey AND other predator horisontally in
32         line with prey OR
33         vertically in line with prey AND other predator vertically in
34         line with prey) // both predators in same alignment with
35         prey
36         AlignWithPrey()
37     if (aligned with prey AND other predator is aligned with prey)
38         MoveToNextPositonCloserToPrey()
39     // SECOND STEP:
40     if (in line with prey AND not adjacent to prey)
41         MoveToNextPositionCloserToPrey()
42     // THIRD STEP:
```

```

38     if (in line with prey and adjacent to prey)
39         {}//Do nothing
40 }
41
42 Method (returns void):AlignWithPrey{
43     verticalDistance = distance from own position to vertically aligned
44         with prey +
45         distance from other predator to horizontally aligned with prey
46     horizontalDistance = distance from own position to horizontally
47         aligned with prey +
48         distance from other predator to vertically aligned with prey
49     if (verticalDistance < horizontalDistance)
50         AlignVerticallyWithPrey()
51     if (horizontalDistance < verticalDistance)
52         AlignHorizontallyWithPrey()
53     if (verticalDistance == horizontalDistance)
54         if (CalculatePositionValue(ownPosition) >
55             CalculatePositionValue(otherPredatorPosition()))
56             AlignHorizontallyWithPrey() // biggest value performs
57             horizontal move
58         else
59             AlignVerticallyWithPrey() // smallest value performs
60             vertical move
61 }
62
63
64 Method (returns integer):CalculatePositionValue{
65     return horizontalPositionValue * 10 + verticalPositionValue // first
66         coordinate * 10 + second coordinate
67 }
68
69
70 Method (returns void):AlignVerticallyWithPrey{
71     if (prey is left of own position)
72         DO: move to adjacent position left of own position
73     if (prey is right of own position)
74         DO: move to adjacent position right of own position
75 }
76
77 Method (returns void):AlignHorizontallyWithPrey{
78     if (prey is above own position)
79         DO: move to adjacent position above own position
80     if (prey is below own position)
81         DO: move to adjacent position below own position
82 }
83
84 Method (returns void):MoveToNextPositionCloserToPrey{
85     if (horizontally aligned with prey AND left of prey)
86         DO: Move to adjacent position right of own position
87     if (horizontally aligned with prey AND right of prey)
88         DO: Move to adjacent position left of own position
89     if (vertically aligned with prey AND above prey)
90         DO: Move to adjacent position below own position
91     if (vertically aligned with prey AND below of prey)
92         DO: Move to adjacent position above own position
93 }

```

Appendix C

Pseudo code for the NXT routine

```
1 // The algorithm makes use of the following existing actions on the NXT
2 // which are indicated in the code through the DO keyword:
3 // DO: Turn left / right
4 // DO: Stear left / right
5 // DO: Drive
6 // DO: Measure light (red, green, blue)
7 // DO: Receive / send messesage from PC
8 // Please be adviced that all variables are global
9
10 // Main routine
11 Module:Main{
12     input = DO: Receive message from PC
13     switch (input)
14         left:
15             DO: Turn left
16             hasTurned = true
17         right:
18             DO: Turn Right
19             hasTurned = true
20         follow:
21             if (countTurns == 0)
22                 leftBlueRightRed = NOT leftBlueRightRed //
23                     used in the FollowLine module
24             countTurns = 0
25             FollowLine
26     }
27     if (hasTurned)
28         turnCount = (turnCount + 1) mod 2
29         hasTurned = false
30     DO: Send message to PC // Completed
31 }
32 Module:FollowLine{
33     stop = false
34     while (stop = false)
35         red, green, blue = DO: Measure light
36         if (blue < threshold) // robot in red field
37             sweep = -1
38         if (red < threshold) // robot in blue field
39             sweep = 1
40         if ((red + blue + green) < blackThreshold) // robot at black
41             point
42             stop = true
43         if (leftBlueRightRed == false) // value set in Main module
44             sweep = sweep * (-1)
```

```
44         switch (sweep)
45             -1:
46                 DO: Stear left
47             0:
48                 DO: Drive
49             1:
50                 DO: Stear right
51 }
```

Appendix D

Unit Test of the Winning Strategy

```
1  using Algorithms.WinningStrategy;
2  using Microsoft.VisualStudio.TestTools.UnitTesting;
3  using CommonComponents;
4
5  namespace Domain_0._2_Test_Project
6  {
7
8
9      /// <summary>
10     /// This is a test class for WinnerPredatorTest and is intended
11     /// to contain all WinnerPredatorTest Unit Tests
12     /// </summary>
13     [TestClass()]
14     public class WinnerPredatorTest
15     {
16         #region Additional test attributes
17         //
18         /// You can use the following additional attributes as you write your tests:
19         //
20         /// Use ClassInitialize to run code before running the first test in the
21         /// class
22         /// [ClassInitialize()]
23         /// public static void MyClassInitialize(TestContext testContext)
24         /// {
25         //
26         /// Use ClassCleanup to run code after all tests in a class have run
27         /// [ClassCleanup()]
28         /// public static void MyClassCleanup()
29         /// {
30         //
31         //
32         /// Use TestInitialize to run code before running each test
33
34
35         /// <summary>
36         /// Initiazises the test.
37         /// </summary>
38         [TestInitialize()]
39         public void MyTestInitialize()
40         {
41             PrivateObject param0 = new PrivateObject(new WinnerPredator(world,
42                 startingPosition));
43             target = new WinnerPredator_Accessor(param0);
44         }
45     }
46 }
```

```

44     }
45     WinnerPredator_Accessor target;
46     /// <summary>
47     /// The World representation
48     /// </summary>
49     World world = new World();
50     Vector startingPosition = new Vector(1, 2);
51     ///
52     /// Use TestCleanup to run code after each test has run
53     /// <summary>
54     /// Cleans up after the test.
55     /// </summary>
56     [TestCleanup()]
57     public void MyTestCleanup()
58     {
59         target = null;
60     }
61     ///
62     #endregion
63
64
65     /// <summary>
66     /// A test for VerticallyInLineWithPrey
67     /// </summary>
68     [TestMethod()]
69     [DeploymentItem("Algorithms.dll")]
70     public void VerticallyInLineWithPreyTest()
71     {
72         bool expected;
73         bool actual;
74
75         for (int preyX = 0; preyX < world.HorizontalSize; preyX++)
76             for (int preyY = 0; preyY < world.VerticalSize; preyY++)
77             {
78                 for (int predatorX = 0; predatorX < world.HorizontalSize; predatorX
79                     ++
80                     )
81                     for (int predatorY = 0; predatorY < world.VerticalSize; predatorY
82                         ++
83                         )
84                         {
85                             /// Re-initialise
86                             Vector position = new Vector(predatorX, predatorY);
87                             PrivateObject param0 = new PrivateObject(new WinnerPredator(
88                                 world, position));
89                             target = new WinnerPredator_Accessor(param0);
90
91                             if (predatorX == preyX)
92                                 expected = true;
93                             else
94                                 expected = false;
95                             actual = target.VerticallyInLineWithPrey(position, new Vector(
96                                 preyX, preyY));
97                             Assert.AreEqual(expected, actual, "Prey_at_(" + preyX + ", " +
98                                 preyY + ") " +
99                                 ",_Predator_at_(" + predatorX + ", " + predatorY + ")");
100                         }
101             }
102     }
103
104     /// <summary>
105     /// A test for RightOfPrey
106     /// </summary>
107     [TestMethod()]
108     [DeploymentItem("Algorithms.dll")]
109     public void RightOfPreyTest()
110     {
111         bool expected;
112         bool actual;

```

```

106
107     for (int preyX = 0; preyX < world.HorizontalSize; preyX++)
108         for (int preyY = 0; preyY < world.VerticalSize; preyY++)
109             {
110                 for (int predatorX = 0; predatorX < world.HorizontalSize; predatorX
111                     ++
112                     for (int predatorY = 0; predatorY < world.VerticalSize; predatorY
113                         ++
114                         {
115                             // Re-initialise
116                             PrivateObject param0 = new PrivateObject(new WinnerPredator(
117                                 world, new Vector(predatorX, predatorY)));
118                             target = new WinnerPredator_Accessor(param0); // TODO:
119                                 Initialize to an appropriate value
120
121                             if (predatorX > preyX)
122                                 expected = true;
123                             else
124                                 expected = false;
125                             actual = target.RightOfPrey(new Vector(preyX, preyY));
126                             Assert.AreEqual(expected, actual, "Prey_at_(" + preyX + ",_" +
127                                 preyY + ")"+
128                                 ",_Predator_at_(" + predatorX + ",_" + predatorY + ")");
129                         }
130                     }
131             }
132
133     /// <summary>
134     ///A test for PositionValue
135     ///</summary>
136     [TestMethod()]
137     [DeploymentItem("Algorithms.dll")]
138     public void PositionValueTest()
139     {
140         Vector position;
141         int expected;
142         int actual;
143
144         for (int x = 0; x < world.HorizontalSize; x++)
145             for (int y = 0; y < world.VerticalSize; y++)
146                 {
147                     position = new Vector(x, y);
148                     expected = x * 10 + y;
149                     actual = target.PositionValue(position);
150                     Assert.AreEqual(expected, actual, "Position:_(" + x + ",_" + y + ")"+
151                         +
152                         ",_Expected:_ " + expected + ",_Actual:_ " + actual);
153                 }
154     }
155
156     /// <summary>
157     ///A test for PerformMove
158     ///</summary>
159     [TestMethod()]
160     public void PerformMoveTest()
161     {
162         Vector otherPredator;
163         Vector ownPosition;
164         Vector[] predatorPositions;
165         Vector preyPosition;
166         Vector expected;
167         Vector actual;
168         PrivateObject param0;
169         WinnerPredator_Accessor target;
170
171         // NOT IN LINE, OTHER PREDATOR IN HORIZONTAL LINE
172         preyPosition = new Vector(0,3);

```

```

167     otherPredator = new Vector (3,3);
168     ownPosition = new Vector(1, 1);
169     predatorPositions = new Vector[2] { ownPosition, otherPredator };
170     expected = new Vector(0, 1);
171     // Re-initialise
172     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
173     target = new WinnerPredator_Accessor(param0);
174     actual = target.PerformMove(predatorPositions, preyPosition);
175     Assert.AreEqual(expected, actual);
176
177     // NOT IN LINE, OTHER PREDATOR IN VERTICAL LINE
178     preyPosition = new Vector(3,2);
179     otherPredator = new Vector (3,3);
180     ownPosition = new Vector(1, 1);
181     predatorPositions = new Vector[2] { ownPosition, otherPredator };
182     expected = new Vector(1, 2);
183     // Re-initialise
184     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
185     target = new WinnerPredator_Accessor(param0);
186     actual = target.PerformMove(predatorPositions, preyPosition);
187     Assert.AreEqual(expected, actual);
188
189     // NOT IN LINE, OTHER PREDATOR NOT IN LINE
190     preyPosition = new Vector(3, 2);
191     otherPredator = new Vector(2, 3);
192     ownPosition = new Vector(1, 1);
193     predatorPositions = new Vector[2] { ownPosition, otherPredator };
194     expected = new Vector(1, 2);
195     // Re-initialise
196     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
197     target = new WinnerPredator_Accessor(param0);
198     actual = target.PerformMove(predatorPositions, preyPosition);
199     Assert.AreEqual(expected, actual);
200
201     // IN LINE WITH PREY AND NOT ADJACENT
202     preyPosition = new Vector(3, 2);
203     otherPredator = new Vector(2, 3);
204     ownPosition = new Vector(3, 0);
205     predatorPositions = new Vector[2] { ownPosition, otherPredator };
206     expected = new Vector(3, 1);
207     // Re-initialise
208     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
209     target = new WinnerPredator_Accessor(param0);
210     actual = target.PerformMove(predatorPositions, preyPosition);
211     Assert.AreEqual(expected, actual);
212
213     // IN LINE WITH PREY AND ADJACENT
214     preyPosition = new Vector(2, 3);
215     otherPredator = new Vector(3, 3);
216     ownPosition = new Vector(0, 3);
217     predatorPositions = new Vector[2] { ownPosition, otherPredator };
218     expected = new Vector(0, 4);
219     // Re-initialise
220     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
221     target = new WinnerPredator_Accessor(param0);
222     actual = target.PerformMove(predatorPositions, preyPosition);
223     Assert.AreEqual(expected, actual);
224
225     // IN LINE WITH PREY, OTHER PREDATOR IN LINE WITH PREY
226
227
228 }
229
230 /// <summary>
231 /// A test for MoveToNextPositionCloserToPrey
232 /// </summary>
233 [TestMethod()]

```

```

234 [DeploymentItem("Algorithms.dll")]
235 public void MoveToNextPositionCloserToPreyTest ()
236 {
237     Vector ownPosition;
238     Vector preyPosition;
239     Vector expected;
240     Vector actual;
241     PrivateObject param0;
242     WinnerPredator_Accessor target;
243
244     // HORIZONTAL AND LEFT OF PREY
245     preyPosition = new Vector(2, 3);
246     ownPosition = new Vector(0, 3);
247     expected = new Vector(1, 3);
248     // Re-initialise
249     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
250     target = new WinnerPredator_Accessor(param0);
251     actual = target.MoveToNextPositionCloserToPrey(preyPosition);
252     Assert.AreEqual(expected, actual);
253
254     // HORIZONTAL AND RIGHT OF PREY
255     preyPosition = new Vector(1, 3);
256     ownPosition = new Vector(3, 3);
257     expected = new Vector(2, 3);
258     // Re-initialise
259     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
260     target = new WinnerPredator_Accessor(param0);
261     actual = target.MoveToNextPositionCloserToPrey(preyPosition);
262     Assert.AreEqual(expected, actual);
263
264     // VERTICAL AND ABOVE PREY
265     preyPosition = new Vector(1, 1);
266     ownPosition = new Vector(1, 3);
267     expected = new Vector(1, 2);
268     // Re-initialise
269     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
270     target = new WinnerPredator_Accessor(param0);
271     actual = target.MoveToNextPositionCloserToPrey(preyPosition);
272     Assert.AreEqual(expected, actual);
273
274     // VERTICAL AND BELOW PREY
275     preyPosition = new Vector(1, 3);
276     ownPosition = new Vector(1, 1);
277     expected = new Vector(1, 2);
278     // Re-initialise
279     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
280     target = new WinnerPredator_Accessor(param0);
281     actual = target.MoveToNextPositionCloserToPrey(preyPosition);
282     Assert.AreEqual(expected, actual);
283 }
284
285 /// <summary>
286 ///A test for LeftOfPrey
287 ///</summary>
288 [TestMethod()]
289 [DeploymentItem("Algorithms.dll")]
290 public void LeftOfPreyTest ()
291 {
292     bool expected;
293     bool actual;
294
295     for (int preyX = 0; preyX < world.HorizontalSize; preyX++)
296         for (int preyY = 0; preyY < world.VerticalSize; preyY++)
297             {
298                 for (int predatorX = 0; predatorX < world.HorizontalSize; predatorX
299                     ++)
```

```

300         ++)
301     {
302         // Re-initialise
303         PrivateObject param0 = new PrivateObject(new WinnerPredator(
304             world, new Vector(predatorX, predatorY));
305         target = new WinnerPredator_Accessor(param0);
306
307         if (predatorX < preyX)
308             expected = true;
309         else
310             expected = false;
311         actual = target.LeftOfPrey(new Vector(preyX, preyY));
312         Assert.AreEqual(expected, actual, "Prey_at_" + preyX + ",_" +
313             preyY + ")" +
314             ",_Predator_at_" + predatorX + ",_" + predatorY + ")");
315     }
316 }
317
318 /// <summary>
319 /// A test for InLineWithPrey
320 /// </summary>
321 [TestMethod()]
322 [DeploymentItem("Algorithms.dll")]
323 public void InLineWithPreyTest()
324 {
325     bool expected;
326     bool actual;
327
328     for (int preyX = 0; preyX < world.HorizontalSize; preyX++)
329         for (int preyY = 0; preyY < world.VerticalSize; preyY++)
330             {
331                 for (int predatorX = 0; predatorX < world.HorizontalSize; predatorX
332                     ++))
333                     for (int predatorY = 0; predatorY < world.VerticalSize; predatorY
334                         ++))
335                         {
336                             // Re-initialise
337                             PrivateObject param0 = new PrivateObject(new WinnerPredator(
338                                 world, new Vector(predatorX, predatorY));
339                             target = new WinnerPredator_Accessor(param0);
340
341                             if (predatorX == preyX || predatorY == preyY)
342                                 expected = true;
343                             else
344                                 expected = false;
345                             actual = target.InLineWithPrey(new Vector(preyX, preyY));
346                             Assert.AreEqual(expected, actual, "Prey_at_" + preyX + ",_" +
347                                 preyY + ")" +
348                                 ",_Predator_at_" + predatorX + ",_" + predatorY + ")");
349                         }
350             }
351 }
352
353 /// <summary>
354 /// A test for HorisontallyInLineWithPrey
355 /// </summary>
356 [TestMethod()]
357 [DeploymentItem("Algorithms.dll")]
358 public void HorisontallyInLineWithPreyTest()
359 {
360     bool expected;
361     bool actual;
362
363     for (int preyX = 0; preyX < world.HorizontalSize; preyX++)
364         for (int preyY = 0; preyY < world.VerticalSize; preyY++)
365             {

```

```

360     for (int predatorX = 0; predatorX < world.HorizontalSize; predatorX
361           ++
362           for (int predatorY = 0; predatorY < world.VerticalSize; predatorY
363                 ++
364                 {
365                     // Re-initialise
366                     Vector position = new Vector(predatorX, predatorY);
367                     PrivateObject param0 = new PrivateObject(new WinnerPredator(
368                         world, position));
369                     target = new WinnerPredator_Accessor(param0);
370
371                     if (predatorY == preyY)
372                         expected = true;
373                     else
374                         expected = false;
375                     actual = target.HorizontallyInLineWithPrey(position, new Vector(
376                         preyX, preyY));
377                     Assert.AreEqual(expected, actual, "Prey_at_(" + preyX + ",_" +
378                         preyY + ") +
379                         ",_Predator_at_(" + predatorX + ",_" + predatorY + ")");
380                 }
381             }
382         }
383     }
384     /// <summary>
385     /// A test for GetOtherPredatorPosition
386     /// </summary>
387     [TestMethod()]
388     [DeploymentItem("Algorithms.dll")]
389     public void GetOtherPredatorPositionTest()
390     {
391         Vector expected;
392         Vector actual;
393         Vector[] positions;
394         Vector otherPredator = new Vector(3, 3);
395
396         positions = new Vector[2] { startingPosition, otherPredator };
397         expected = otherPredator;
398         actual = target.GetOtherPredatorPosition(positions);
399         Assert.AreEqual(expected, actual);
400
401         positions = new Vector[2] { otherPredator, startingPosition };
402         expected = otherPredator;
403         actual = target.GetOtherPredatorPosition(positions);
404         Assert.AreEqual(expected, actual);
405     }
406     /// <summary>
407     /// A test for BelowPrey
408     /// </summary>
409     [TestMethod()]
410     [DeploymentItem("Algorithms.dll")]
411     public void BelowPreyTest()
412     {
413         bool expected;
414         bool actual;
415
416         for (int preyX = 0; preyX < world.HorizontalSize; preyX++)
417             for (int preyY = 0; preyY < world.VerticalSize; preyY++)
418                 {
419                     for (int predatorX = 0; predatorX < world.HorizontalSize; predatorX
420                           ++
421                           for (int predatorY = 0; predatorY < world.VerticalSize; predatorY
422                                 ++
423                                 {
424                                     // Re-initialise
425                                     PrivateObject param0 = new PrivateObject(new WinnerPredator(

```

```

420         world, new Vector(predatorX, predatorY));
421     target = new WinnerPredator_Accessor(param0);
422
423     if (predatorY < preyY)
424         expected = true;
425     else
426         expected = false;
427     actual = target.BelowPrey(new Vector(preYX, preyY));
428     Assert.AreEqual(expected, actual, "Prey_at_" + preyX + "," +
429         preyY + ")" +
430         ",_Predator_at_" + predatorX + "," + predatorY + ");
431     }
432 }
433
434 /// <summary>
435 ///A test for AlignWithPrey
436 ///</summary>
437 [TestMethod()]
438 [DeploymentItem("Algorithms.dll")]
439 public void AlignWithPreyTest()
440 {
441     Vector otherPredator;
442     Vector ownPosition;
443     Vector preyPosition;
444     Vector[] predatorPositions;
445     Vector expected;
446     Vector actual;
447     PrivateObject param0;
448     WinnerPredator_Accessor target;
449
450     // VERTICAL < HORIZONTAL
451     preyPosition = new Vector(3, 2);
452     otherPredator = new Vector(2, 3);
453     ownPosition = new Vector(1, 1);
454     predatorPositions = new Vector[2] { startingPosition, otherPredator };
455     expected = new Vector(1, 2);
456     // Re-initialise
457     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
458     target = new WinnerPredator_Accessor(param0);
459     actual = target.AlignWithPrey(otherPredator, preyPosition);
460     Assert.AreEqual(expected, actual);
461
462     // VERTICAL > HORIZONTAL
463     preyPosition = new Vector(3, 2);
464     otherPredator = new Vector(1, 1);
465     ownPosition = new Vector(2, 3);
466     predatorPositions = new Vector[2] { startingPosition, otherPredator };
467     expected = new Vector(3, 3);
468     // Re-initialise
469     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
470     target = new WinnerPredator_Accessor(param0);
471     actual = target.AlignWithPrey(otherPredator, preyPosition);
472     Assert.AreEqual(expected, actual);
473
474     // VERTICAL == HORIZONTAL (BIGGER VALUE)
475     preyPosition = new Vector(2, 2);
476     otherPredator = new Vector(1, 0);
477     ownPosition = new Vector(3, 0);
478     predatorPositions = new Vector[2] { startingPosition, otherPredator };
479     expected = new Vector(3, 1);
480     // Re-initialise
481     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
482     target = new WinnerPredator_Accessor(param0);
483     actual = target.AlignWithPrey(otherPredator, preyPosition);
484     Assert.AreEqual(expected, actual);

```

```

485 // VERTICAL == HORIZONTAL (LOWER VALUE)
486 preyPosition = new Vector(2, 2);
487 otherPredator = new Vector(3, 0);
488 ownPosition = new Vector(1, 0);
489 predatorPositions = new Vector[2] { startingPosition, otherPredator };
490 expected = new Vector(2, 0);
491 // Re-initialise
492 param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
493 target = new WinnerPredator_Accessor(param0);
494 actual = target.AlignWithPrey(otherPredator, preyPosition);
495 Assert.AreEqual(expected, actual);
496 }
497
498 /// <summary>
499 ///A test for AlignVerticallyWithPrey
500 ///</summary>
501 [TestMethod()]
502 [DeploymentItem("Algorithms.dll")]
503 public void AlignVerticallyWithPreyTest ()
504 {
505     Vector ownPosition;
506     Vector preyPosition;
507     Vector expected;
508     Vector actual;
509     PrivateObject param0;
510     WinnerPredator_Accessor target;
511
512     // LEFT OF PREY
513     preyPosition = new Vector(2, 3);
514     ownPosition = new Vector(0, 2);
515     expected = new Vector(1, 2);
516     // Re-initialise
517     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
518     target = new WinnerPredator_Accessor(param0);
519     actual = target.AlignVerticallyWithPrey(preyPosition);
520     Assert.AreEqual(expected, actual);
521
522     // RIGHT OF PREY
523     preyPosition = new Vector(1, 3);
524     ownPosition = new Vector(3, 2);
525     expected = new Vector(2, 2);
526     // Re-initialise
527     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
528     target = new WinnerPredator_Accessor(param0);
529     actual = target.AlignVerticallyWithPrey(preyPosition);
530     Assert.AreEqual(expected, actual);
531 }
532
533 /// <summary>
534 ///A test for AlignHorizontallyWithPrey
535 ///</summary>
536 [TestMethod()]
537 [DeploymentItem("Algorithms.dll")]
538 public void AlignHorizontallyWithPreyTest ()
539 {
540     Vector ownPosition;
541     Vector preyPosition;
542     Vector expected;
543     Vector actual;
544     PrivateObject param0;
545     WinnerPredator_Accessor target;
546
547     // BELOW PREY
548     preyPosition = new Vector(2, 3);
549     ownPosition = new Vector(0, 1);
550     expected = new Vector(0, 2);
551     // Re-initialise

```

```

552     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
553     target = new WinnerPredator_Accessor(param0);
554     actual = target.AlignHorisontallyWithPrey(prePosition);
555     Assert.AreEqual(expected, actual);
556
557     // ABOVE PREY
558     preyPosition = new Vector(2, 0);
559     ownPosition = new Vector(0, 3);
560     expected = new Vector(0, 2);
561     // Re-initialise
562     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
563     target = new WinnerPredator_Accessor(param0);
564     actual = target.AlignHorisontallyWithPrey(prePosition);
565     Assert.AreEqual(expected, actual);
566 }
567
568 /// <summary>
569 ///A test for AdjacentToPrey
570 ///</summary>
571 [TestMethod()]
572 [DeploymentItem("Algorithms.dll")]
573 public void AdjacentToPreyTest()
574 {
575     Vector ownPosition;
576     Vector preyPosition;
577     bool expected;
578     bool actual;
579     PrivateObject param0;
580     WinnerPredator_Accessor target;
581
582     // BELOW PREY
583     preyPosition = new Vector(2, 3);
584     ownPosition = new Vector(2, 2);
585     expected = true;
586     // Re-initialise
587     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
588     target = new WinnerPredator_Accessor(param0);
589     actual = target.AdjacentToPrey(prePosition);
590     Assert.AreEqual(expected, actual);
591
592     // ABOVE PREY
593     preyPosition = new Vector(2, 2);
594     ownPosition = new Vector(2, 3);
595     expected = true;
596     // Re-initialise
597     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
598     target = new WinnerPredator_Accessor(param0);
599     actual = target.AdjacentToPrey(prePosition);
600     Assert.AreEqual(expected, actual);
601
602     // LEFT OF PREY
603     preyPosition = new Vector(2, 3);
604     ownPosition = new Vector(1, 3);
605     expected = true;
606     // Re-initialise
607     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
608     target = new WinnerPredator_Accessor(param0);
609     actual = target.AdjacentToPrey(prePosition);
610     Assert.AreEqual(expected, actual);
611
612     // RIGHT OF PREY
613     preyPosition = new Vector(2, 3);
614     ownPosition = new Vector(3, 3);
615     expected = true;
616     // Re-initialise
617     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
618     target = new WinnerPredator_Accessor(param0);

```

```

619     actual = target.AdjacentToPrey(preYPosition);
620     Assert.AreEqual(expected, actual);
621
622     // NOT ADJACENT
623     preyPosition = new Vector(3, 3);
624     ownPosition = new Vector(1, 1);
625     expected = false;
626     // Re-initialise
627     param0 = new PrivateObject(new WinnerPredator(world, ownPosition));
628     target = new WinnerPredator_Accessor(param0);
629     actual = target.AdjacentToPrey(preYPosition);
630     Assert.AreEqual(expected, actual);
631 }
632
633 /// <summary>
634 /// A test for AbovePrey
635 /// </summary>
636 [TestMethod()]
637 [DeploymentItem("Algorithms.dll")]
638 public void AbovePreyTest()
639 {
640     bool expected;
641     bool actual;
642
643     for (int preyX = 0; preyX < world.HorizontalSize; preyX++)
644         for (int preyY = 0; preyY < world.VerticalSize; preyY++)
645             {
646                 for (int predatorX = 0; predatorX < world.HorizontalSize; predatorX
647                     ++
648                     for (int predatorY = 0; predatorY < world.VerticalSize; predatorY
649                         ++
650                         {
651                             // Re-initialise
652                             PrivateObject param0 = new PrivateObject(new WinnerPredator(
653                                 world, new Vector(predatorX, predatorY)));
654                             target = new WinnerPredator_Accessor(param0);
655
656                             if (predatorY > preyY)
657                                 expected = true;
658                             else
659                                 expected = false;
660                             actual = target.AbovePrey(new Vector(preYX, preyY));
661                             Assert.AreEqual(expected, actual, "Prey_at_" + preyX + ",_" +
662                                 preyY + ")" +
663                                 ",_Predator_at_" + predatorX + ",_" + predatorY + ")");
664                         }
665                     }
666             }
667 }
668
669 /// <summary>
670 /// A test for WinnerPredator Constructor
671 /// </summary>
672 [TestMethod()]
673 public void WinnerPredatorConstructorTest()
674 {
675     World map = world;
676     Vector pos = startingPosition;
677     WinnerPredator target = new WinnerPredator(map, pos);
678     Assert.AreEqual(target.Vector, startingPosition);
679 }
680 }

```