AALBORG UNIVERSITY

Jetpack Attack:

Artificial Intelligence for Project Hoshimi

SW10 - Spring 2008

Project Group: d632a

AALBORG UNIVERSITY

Department of Computer Science
Selma Lagerlöfsvej 300
http://www.cs.aau.dk

Title:
Jetpack Attack: AI for Project Hoshimi.

Theme:
Machine Intelligence

Semester:
SW10, 1st of February - 4th of June 2008

Group:
d632a

Members:
Henrik Kronborg Andersen
Morten Krog Sneftrup Pedersen

Supervisor:
Yifeng Zeng

Copies: π^n

Report - pages: e^n

Appendices: 0

DVD: 1

Total pages: $e^n + \infty$

Abstract:

This report describes the process of creating a competitive AI system named *Jetpack Attack* for the programming game Project Hoshimi, using various Machine Intelligence techniques. The report is a study of implementing a complete multi-agent AI, but has emphasis on two specific parts of the game; finding a starting point, and classifying the mission.
The rules of Project Hoshimi are described, followed by the design of the AI. A range of different AI techniques are then described, and the useful ones are implemented, and a new method for finding the landing point is developed.
The final implementation is then tested with various experiments, including a submission to a world-wide competition. The results of these experiments are described and the report concludes on the project with an evaluation of the various results.

Preface

The following report is written during the spring of 2008 by two Software Engineering students, at the Department of Computer Science at Aalborg University.

When the words *we* and *our* are used, it refers to *the authors* of this report and *he* refers to *he/she*.

When code is presented, it may differ from the actual source code. It may have been modified and have had some details removed to make it fit into the report. This has been done to heighten the legibility of the code, and to focus on essential functionality.

The first time an abbreviation is used, the entire word or sentence is written, followed by the abbreviation in parentheses. Throughout the rest of the report, the abbreviation is used. Abbreviations and their meanings are located in Appendix A.

It is expected that the reader has basic knowledge of software engineering and machine intelligence.

Henrik Andersen

Morten Pedersen

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Artificial Intelligence (AI) in games is becoming more and more advanced, along with receiving more and more attention. Competitions like *DreamBuildPlay*[4] and *Imagine Cup*[9] emphasizes AI, and game developers are putting a lot of effort into creating very sophisticated AI in their games. One of the many challenges of implementing a good AI, is making it perform well, despite involving a lot of complex calculations.

One specific type of AI, is the multi-agent system, which has a number of autonomous agents, interacting with each other, which must be able to solve complex tasks. This project is a general study of how to implement such a multi-agent AI for a game. In the chosen game, main focus is on developing a method for selecting a starting point, and classification of a given mission, and using that classification to decide what overall strategy to use. For both of these points, and advanced solution will be built, that should improve the overall usefulness of the AI.

In order to avoid developing a full game, the programming game Project Hoshimi[2], which is part of the Imagine Cup competition, has been chosen. Project Hoshimi is a game where the player creates an AI to control a fleet of robots, that have to solve different objectives, in order to get the highest score. This game has been chosen for two reasons. Firstly, all the necessary Software Development Kits (SDK) and tools are freely available, and should be of high quality, which will make it easier to focus on developing the actual AI. Secondly, because it is part of a well-known and respected competition, several qualified opponents should be available for us to test our AI against, which will help us evaluate our efforts. Project Hoshimi also presents a challenge, performance wise, since it is turn-based and thus enforces that actions must be completed within the given time limit of a turn.

The obvious goal is to create a competitive AI, that will rank among the top players in the competition. The optimal goal would be a first place, but since many of the opponents we will be competing against, have participated in the competition several years in a row, first place is unlikely. Furthermore, the goal is to prove that our solutions to the two points of focus improve the overall AI, and that they are general techniques that can be used in other contexts than Project Hoshimi.

This report covers the development of an AI for Project Hoshimi, named *Jetpack Attack*. The techniques used for the implementation along with related work will be presented, and finally the AI will be tested and evaluated, with regards to whether or not it fulfills the goals set in the preceding paragraph. The remainder of this report, is structured as follows:

- **Chapter 2 - Project Hoshimi:** describes the rules of Project Hoshimi.
- **Chapter 3 - AI Design:** describes the common strategies used by other players, and the design of our AI.
- **Chapter 4 - AI Techniques:** describes the techniques used for the different parts of the AI.
- **Chapter 5 - Implementation:** describes the implementation of *Jetpack Attack*.
- **Chapter 6 - Experiments:** describes various experiments performed to test the results of this project.
- **Chapter 7 - Conclusion:** evaluates and concludes upon the project.

Chapter 2

Project Hoshimi

This chapter describes the main focus of this report, i.e. Project Hoshimi[2], which is a part of a competition called the Imagine Cup[9]. Project Hoshimi is a programming competition, where the purpose is to create an AI, which will play against AIs, created by other participants in the competition.

The story behind Project Hoshimi, is that the Earth has become over-populated and extremely polluted. The pollution is especially the result of the actions of a man named Pierre, who builds polluting factories at different locations all over the world. To help heal the Earth, scientists have discovered a gas called OXY gas, which reduces pollution when injected into special locations on the Earth. These locations are called Hoshimi points. Apart from healing the Earth, the source of the pollution must also be removed, and therefore Pierre's factories must be destroyed.

The role of a participator in the Project Hoshimi competition, is to create an AI, which helps stop pollution by destroying Pierre's factories, and injecting OXY gas into Hoshimi points. To do this, a number of bots are available, each of which have different strengths and weaknesses, and different abilities. These bots are described in Section 2.1. When the AI is ready for deployment, it is given a map and a number of objectives to solve. The combinations of a map and the objectives, are called missions, and these are described in Section 2.2. When given a mission, the first task that must be performed, is selecting a landing point, which can be anywhere in the map. This location is where all bots are initially built, so therefore choosing the landing point, must be done with great care.

The competition part of Project Hoshimi, consists of pitting your AI against an AI created by another competitor. This is done by giving both AIs the same mission, and thus deploying them simultaneously in the same map, with the same objectives.

2.1 Bots

There are several types of bots in Project Hoshimi, all of which are built and controlled by a single bot, called the AI. The different bots which can be built, all have different abilities and characteristics which makes them more or less suited to perform a given task. It is not possible to have more than 40 bots at any point during the game. The different types of bots, and their graphical representation in the game, are the following:

- **- AI:** builds and controls the other bots.
- **- Container:** used for collecting OXY gas and transporting it to Hoshimi points.
- **- Collector:** can also collect and transport OXY gas, and is furthermore able to defend itself.
- **- Explorer:** fast bot which ignores slow areas on the map, and can see far, but has a limited amount of health.
- **- Needle:** non-moving bot, used for injecting OXY gas into the Earth.
- **- Blocker:** stationary bot, which slows opponent's bots down in an area around itself.
- **- Wall:** another non-moving bot, which completely stops the opponent's bots from passing.
- **- LPCreator:** used for creating a second landing point, where new bots can be built.

As mentioned previously, all bots are initially built at the selected landing point, but with the *LPCreator*, it is possible to open a new landing point anywhere on the map, and then have the option of using this point for building bots. However, the *Needle*, *Wall* and *Blocker* bots are not built at the new landing point nor the old one for that matter, but instead are always built at the location of the AI bot, since these bots can not move. Once an *LPCreator* has opened the landing point, the bot will only be alive for 500 more turns, after which the new landing point will close, and it will not be possible to build an *LPCreator* again during the game.

2.2 Missions

As mentioned previously, a mission in Project Hoshimi consists of a map, and a number of objectives. The map consists of different terrain types all of which have different attributes. Water is not passable and is therefore used for outlining the map and creating natural obstacles, such as rivers, lakes and fjords. Other than that, there are two types of terrain which makes bots move slow and very slow respectively, and one type which makes the bots move faster. A sample map can be seen in Figure 2.1.

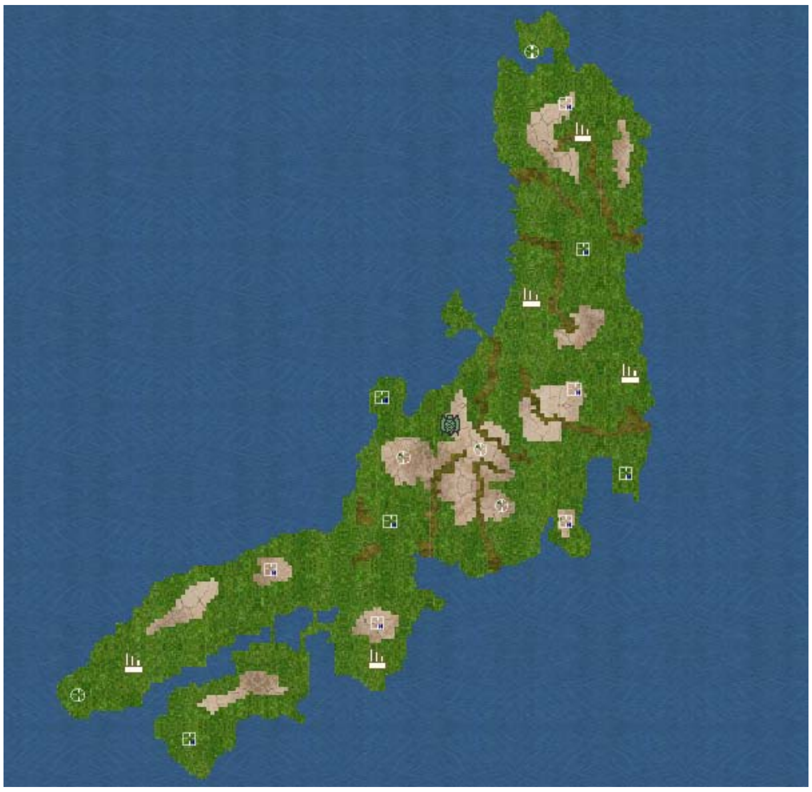

Figure 2.1: Sample Map

The symbols on the map, signify different things, i.e. the circles (⊕) are OXY gas locations, the squares (▣) are Hoshimi points, and the white factories (▥), are indeed factories. The light-brown areas make the bots move slowly, and the dark-brown areas make the bots move very slowly.

To win a mission, the AI must have a higher score than the opponent's AI, when the game is over, which is when 1500 turns have passed. It is possible to score points by completing objectives and by injecting OXY gas in to the Earth, at Hoshimi points. After selecting a landing point, the AI must attempt to defeat the opponent. This is done using the different bots, described in Section 2.1. The following section describes the different types of objectives possible to encounter in a mission, and how they can be completed. It should be noted, that in the case of the two players having the same number of points at the end of the game, there are four rules which decides the outcome of the game:

1. Number of objectives done
2. Needle bots placed on Hoshimi points
3. Number of Needle bots
4. Number of bots

This means that if two players have the same score, the one with the most completed objectives will be the winner. If this number also is equal, number two in the list is used to determine a winner, and so on.

2.2.1 Objectives

There are five different types of objectives in Project Hoshimi, all of which gives a number of points and most of them can appear more than once in any given mission. Some objectives are of a more passive nature while others are more active. Apart from the five objectives in the official rules, we have chosen to add a sixth one, which is of high importance, when it comes to scoring points, and should therefore be treated as an objective in itself.

2.2.1.1 Passive Objectives

The passive objectives are ones which do not require specific actions of the AI, but instead are achieved, if not automatically, then at least without making explicit decisions to achieve them. Achieving them only for the sake of doing so is therefore not as important as the active objectives, but nevertheless can be crucial to winning the game. For example, the *AI Alive* objective aims to keep the AI bot alive for a certain number of turns. If this goal is reached, a number of points is awarded to the player. Achieving this objective also makes a lot of sense in itself, since if the AI bot dies, all the other bots will finish their current action and not be able to do anything after that, and thus leaving the opponent free to do as he chooses. The last passive objective, is the *Score* objective, which is accomplished by having a certain score by a specific turn. Again, this objective makes sense, since it is the goal of the game to get the highest possible score.

2.2.1.2 Active Objectives

The active objectives are ones which can not be completed, without explicitly taken them into account, when devising a strategy. There are three of these, plus the one we have added. There are two that deal with navigation, i.e. the *Navigation* objective and the *Unique Navigation* objective. The first one lists a number of locations in the map, and to achieve the objective, all these locations must be visited, before the game ends. This can be done with any number of bots. The *Unique Navigation* objective also has a list of locations which must be visited. The only difference from the *Navigation* objective, is that in this objective all the locations must be visited by one specific bot. It should be noted that both players can achieve these objectives, and receive full points for them. There can be any number of both *Navigation* objectives and *Unique Navigation* objectives in a mission. Both navigation objectives can also require that it is completed by a specific type of bot, e.g. a *Collector* or an *Explorer*.

Another active objective is the *Factory* objective, where the goal simply is to destroy a number of factories. This number will always be more than half of the total number of factories, so only one player can achieve the objective. Therefore it is important to quickly accomplish the objective, before the opponent does. Destroying factories can also influence the completion of other objectives, since they have the ability to shoot and damage bots, and thus preventing them from passing by.

The last active objective, is the one we have added and named the *Hoshimi* Objective. As mentioned previously, apart from achieving objectives, scoring points can also be done by collecting OXY gas, and injecting it into the Earth at Hoshimi points. Since achieving the highest score is crucial to winning a game, it has been deemed necessary to treat the process of collecting the OXY gas, transporting it to Hoshimi points, and injecting it into the earth, as it was an objective. Therefore it will be considered an objective along with the others, for all intent and purposes.

This concludes the description of Project Hoshimi, and the different resources available when creating an AI for the competition.

Chapter 3

AI Design

This chapter describes the initial design of the AI for *Jetpack Attack*. This includes the behavior of the different bots, along with how they are controlled, and other features which helps in scoring points in the game. To both gain an understanding of the opponents we might face, and also serve as an inspiration for *Jetpack Attack*, the following section surveys AIs developed by other players. After that, Section 3.2 describes the overall behavior of *Jetpack Attack* and some of the features used to create that behavior.

3.1 Common Strategies

This section describes some of the common strategies for solving active objectives and for selecting the initial landing point. The strategies have been found by examining replays of *Small Competition* games from the Project Hoshimi website, and they are used to gain knowledge of how players complete the objectives and position their landing point. Focus is on active objectives, as they are the only objectives that require non-trivial strategies. This information is used when the strategy for *Jetpack Attack* is created.

3.1.1 Landing Point

Picking a landing point is one of the most important aspects in Project Hoshimi. It is also one of the hardest aspects to analyze from replays, as a players most likely take several factors into account when selecting a landing point, that cannot directly be seen from replays. However, using replays will give some indication of what players take into account when computing their

Player	Factories	Nav	UNav	Hoshimi	OXY	Notes
G3	3	3	1	1	1	On OXY
KTropic	4	0	2	5	1	On Hoshimi
NanoTim	5	1	1	6	1	On Hoshimi
NoOne	3	0	1	5	0	On Hoshimi

Table 3.1: Landing Points in Small Competition 2 Final Round

landing point.

Table 3.1 lists the information collected from the replays of the *Small Competition 2* final round. The information was gathered by finding all objective points in a radius of 30 from the landing point.

Although Table 3.1 is only based on four players, it seems to indicate that players tend to place their landing point close to hoshimi points, OXY points and factories. Considering that hoshimi points only can be occupied by one *Needle* bot at a time, ensuring that you start close to these can be essential. In the map used for *Small Competition 2* factories are located right next to hoshimi points, which explains why the landing point of every player is located next to factories. It is interesting to note that the landing point of every player is on top of a hoshimi or OXY point. This means that, in the case of hoshimi points, the player can build a *Needle* bot without having to move the *AI* bot.

3.1.2 Navigation Objectives

There are two common strategies for solving navigation objectives. One is to simply create a bot for every navigation point, thereby traversing every point in one go. Another strategy, that seems to be the most widely used, is to create a certain number of bots, and then make them traverse the different points one at a time. Along with the two different strategies, there is also a variance in the type of bot used when moving to objective points, and how the player prioritize navigation objectives all together.

Table 3.2 lists the different strategies used by the players in *Small Competition 2*. *Unit count* is an estimate on how many units are assigned to navigation objectives, and *Priority* is an estimated priority, based on how long it takes for the AI to start focusing on navigation objectives.

The data has been gathered by watching replays, and therefore might not be entirely accurate. However, it shows that some players seem to use

Player	**Units used**	**Unit count**	**Priority**
G3	Both	Few units	Low
NoOne	Both	Several units	Medium
KTropin	Collectors	Several units	Medium
NanoTim	Both	Several units	Low
AravindaDP_SC2	Explorers	Few units	Medium
BHKongMing	Both	Several units	Medium
DGAP_team	Both	Few units	Medium
Bonnet	Collectors	Many units	High
Zephyr	Explorers	Few units	Medium
Darkspy	Collectors	Few units	Low
Sphinx	Both	Few units	Low

Table 3.2: Navigation Strategy in Small Competition 2

one type of bots consistently, either *Explorer* or *Collector* bots, except when the navigation objective specifically specifies which bot type must be used. This makes it hard to draw a specific conclusion regarding which type to use. One thing that can be concluded, is that every single player tries to solve the navigation objectives.

3.1.3 Unique Navigation Objectives

Accomplishing unique navigation objectives is significantly harder than the navigation objectives. The unique-navigation objectives require traveling more by one bot than navigation objectives, which means more time for the bot be destroyed. For this reason, it seems like most players use *Explorer* bots to accomplish them, due to their higher movement speed. This leads to the question of whether it pays off to actually try an achieve these.

Table 3.3 lists different statistics from the 12 games in the final round in *Small Competition 2*. The *Nav.* and *UNav.* columns list how many navigation objectives and unique-navigation objectives each player completed out of the maximum possible. The *Score* column is the score at the end of the game, and *AI Alive* list whether the players AI was alive at the end of the game or not.

From the statistics in Table 3.3, it seems fair to conclude that winning the game is related to the amount of navigation and unique navigation objectives completed. It also becomes visible that solving unique objectives is far from trivial, and that even though most players try to complete them, they rarely

Game	Nav. (4)	UNav. (3)	Score	AI Alive	Winner
G3, KTropin	1, 3	0, 1	1200, 4495	no, yes	KTropin
G3, NanoTim	2, 3	0, 1	2240, 4490	no, yes	NanoTim
G3, NoOne	2, 2	0, 1	2120, 4020	yes, yes	NoOne
KTropin, G3	3, 3	0, 1	4505, 2200	yes, no	KTropin
KTropin, NanoTim	1, 4	1, 2	2680, 5680	no, yes	NanoTim
KTropin, NoOne	0, 0	0, 0	1060, 5	no, no	KTropin
NanoTim, G3	2, 3	0, 0	3100, 2940	no, yes	NanoTim
NanoTim, KTropin	4, 1	2, 0	6350, 2065	yes, no	NanoTim
NanoTim, NoOne	4, 3	3, 0	4765, 2640	yes, no	NanoTim
NoOne, G3	4, 2	1, 0	4405, 2340	yes, yes	NoOne
NoOne, KTropin	0, 0	0, 0	5, 1060	no, no	KTropin
NoOne, NanoTim	3, 4	0, 3	2425, 4585	no, yes	NanoTim

Table 3.3: Final Round Statistics in Small Competition 2

succeed. This should be taken into account when deciding how much to prioritize them.

3.1.4 Factory

Factory objectives seem to be prioritized high by the players as they often attack factories within the first 200 turns. However, it seems like most of the players in *Small Competition 2* are not completely aware of how scoring works for factory objectives, as three of four players consistently attacks more than one factory. It should be noted that the maps used for this survey, uses an old version of the factory objective, where all factories had to be destroyed to gain points. However, it was enough to just kill on factory and let the other player destroy the others to gain points for completion.

When attacking factories, players simply send *Collector* bots to the factories, and attack out of range from the factories. The amount of units sent to destroy the factories varies from player to player.

3.1.5 Hoshimi

The strategy for accomplishing hoshimi objectives require two steps; build a *Needle* bot on a hoshimi point, and collect OXY gas and deposit it in the *Needle*.

The players in the final round of *Small Competition 2* use three different strategies for accomplishing OXY gas objectives. The first player selects his landing point on a hoshimi point, builds an *LPCreator* bot as the first bot, sends the *LPCreator* bot to the nearest OXY point and opens a landing point there. After building a *Needle* bot, he starts building *Container* bots at the newly open landing point, fills them, and sends them towards the *Needle* bots and fills the *Needle*. The second and third player creates several *Container* bots and the initial landing point, sends them to the nearest OXY point to collect gas, while building *Needle* bots. The last player selects a landing point on a OXY point, and builds *Container* bots when the first *Needle* bot has been build.

This concludes the survey of the common strategies used by other players in Project Hoshimi. The following describes how *Jetpack Attack* will behave, inspired by the previous sections.

3.2 Jetpack Attack Behavior

This section describes the overall behavior of the bots in *Jetpack Attack*, along with some of the techniques which will be used to implement this behavior. The first thing that must be done when the game starts, is picking a landing point, and as described previously, this is very important for how well the strategy works. There are many factors that go into deciding a good landing point, such as the number of, and locations of the hoshimi points, OXY points and factories, and the layout of the terrain. Our strategy is to do an initial analysis of the given mission, to determine which factors are the most important. This will be based on a number of things, e.g. the total number of factories and how many must be killed to fulfil an objective, how many hoshimi points there are, and how far they are apart, etc. When this analysis is done, and it is determined that, for instance, hoshimi points are the most important point-giver in the game, a landing point will be selected which enables quick access to as many hoshimi points as possible. How the mission will be analyzed, and the landing point selected, is described in Chapter 4. Once a landing point has been chosen, the next goal is achieving objectives, and scoring as many points as possible. To do this, a number of different features are designed, and these are described in the following sections.

3.2.1 Protecting the AI

One of the most important things to do in a game is to keep the *AI* bot alive. To help ensure the security of the *AI* bot, first an *Explorer* bot is built, called the *AIScout*. Because the *Explorer* is capable of seeing further than any other bot, the *AIScout* is useful for extending the viewing distance of the *AI* bot, and thus helping it better foresee what will happen in future turns. The *AIScout* will follow the *AI* constantly. Since neither the *AI* nor the *AIScout* are capable of defending themselves, a *Collector* bot called a *Bodyguard* will also follow the *AI* around, providing attack power, for when enemies come too close. Despite of the *Bodyguard*, the *AI* can still be killed. Therefore a stationary *Needle* bot, the *DefenseNeedle*, is introduced. This type of bot, has the advantage of being built at the location of the *AI*, instead of at the landing point, and it has a shield which protects all nearby bots while it is up. So when an enemy is spotted near the *AI*, it stops and builds a *DefenseNeedle*. If the shield of the *DefenseNeedle* becomes low, it will auto-destruct, and another one will be built immediately. The *DefenseNeedle* also has the added bonus of being able to shoot, and can therefore help kill the enemies faster. By using the *AIScout*, the *Bodyguard* and the *DefenseNeedle*, the *AI* should be very hard to kill, and capable of defending itself efficiently.

3.2.2 Navigation Objectives

Now that the *AI* is protected, we will focus on completing objectives. For the unique navigation objectives, *Explorer* bots, called *Scouts* will be used, unless the objective calls for a specific bot to be used. These bots have the advantage of being faster than all other bots, and this is often necessary to reach all goals of a unique navigation objective before the end of the game. For regular navigation objectives, *Collector* bots called *Destroyers* will be used, as it is possible to send one of these out to each goal of several different navigation objectives, and they are able to defend themselves, if they run into enemies along the way. Again, the *Destroyers* will only be used, if the objective does not call for a specific bot.

3.2.3 Hoshimi Objectives

To collect OXY gas, *Container* bots, named *Transporters* will be used, and these bots will also be able to complete navigation objectives, if the objective itself stipulates this. To inject the gas into the earth at hoshimi points, *Needle* bots called *InjectorNeedles* will be used. The *Transporters* will collect the gas and then either move to an available *InjectorNeedle* placed on a hoshimi point, or if none are available, move to the current target of the *AI*, since this will be the next place an *InjectorNeedle* will be built.

3.2.4 Hunter Bots

The last type of bot being used in *Jetpack Attack*, is another *Collector* bot, called the *Hunter*. At all times during a game, there will be five of these bots, and their responsibility is to, as the name suggests, hunt the enemy bots. To do this they will randomly patrol the map, and attack all enemies it encounters. If another bot spots the enemy *AI*, all *Hunters* will immediately give up their current target, and move to where the *AI* was spotted. Killing the enemy *AI* would give our team a very large advantage since, as mentioned previously, if your *AI* dies, you will no be able to perform any actions for the rest of the game, and thus leaving your opponent to do as he likes.

3.2.5 Miscellaneous Techniques

Apart from the bots and techniques described in the previous sections, there are also a few other techniques which are valuable for gaining an advantage over the opponent. First of all, the bots which are not able to defend themselves (such as the *Scout* and the *Transporter*), must be able to call for help, when an enemy is spotted, which can potentially hurt the bot. When a call for help has been made, a *Destroyer* bot must come to the aid of the bot in need. This can either be a new bot, or one which has already been built.

Another thing which can give an edge over the opponents, is utilizing the way a bot attacks. As described previously, all bots capable of attacking, has a characteristic called *DefenseDistance*, which is how far they can shoot. When they shoot at a point, damage will be dealt in a circle around that point. This is illustrated in Figure 3.1.

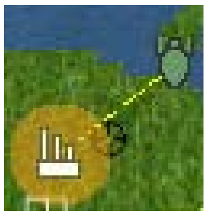

Figure 3.1: Standard way of attacking.

The strategies used by the other players, described in Section 3.1, all use the standard way of attacking other players, i.e. when the other player's bot is spotted, the bot will fire at the location of the other bot. This is actually less than optimal, since other bots are often spotted before they are within the *DefenseDistance* of the spotter. *Jetpack Attack* will use this fact to gain an advantage over other players. By taking into account that damage is dealt in a circle around the point being attacked, it is possible to shoot at enemies outside of a bots *DefenseDistance*, as can be seen in Figure 3.2.

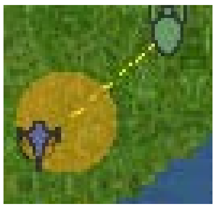

Figure 3.2: Jetpack Attack way of attacking.

By doing this, it is possible to kill an enemy with the same capabilities as our own bot, before he reaches us, with his attack.

As described above, *Jetpack Attack* will use multiple bots, with different abilities, strengths and weaknesses, to accomplish objectives and score points. To keep track of the different bots and objectives, a central control unit is used, and the concept of *Tasks* is introduced. Since every way of scoring points basically comes down to moving to a location and performing some action, tasks are very generic entities which contains a number of targets and the type of bot needed to complete the task. To handle the delegation of tasks, the central control unit is needed, to keep track of the given mission, and which tasks are of the highest priority.

Chapter 4

AI Techniques

This chapter describes some of the techniques used to implement the main features of *Jetpack Attack*. As described previously, the two main focus points of this project are finding a good landing point, and the initial analysis of the mission. Apart from these two points, a description of how bots will behave is presented.

4.1 Related Work

This section describes two techniques for finding a landing point in Project Hoshimi. Because Project Hoshimi is a competition, most of the information about how the competitors build their AI is kept secret. The only information available is the official documentation. The two techniques described here are taken from this documentation [11].

4.1.1 Barycenter

This technique uses the barycenter of the objectives in the map to find a landing point. It is one of the approaches to find a landing point provided by Project Hoshimi.

The barycenter is defined as the center of mass of one or more objects. In this case, it is used to provide a weighted average of the different objectives in the map. Calculating the landing point is very simple using this approach. First, three average points are calculated from every hoshimi point, OXY point and factory. The landing point is then found by calculating a weighted average of those three points.

Although the landing point can be found very fast using this approach, the landing point found, is rarely in a desirable location, and does not take the terrain into consideration.

4.1.2 Gaussian Function

This techniques works by rating every point in the map using a two-dimensional Gaussian function, and picking the highest value point. This section is based on [11] and [13].

By rating every point in the map, finding the landing point is done simply by picking the highest rated point. In order to rate the maps, a two-dimensional Gaussian function

$$f(x, y) = Ae^{-\left(\frac{(x-x_O)^2}{2\sigma_x^2}+\frac{(y-y_O)^2}{2\sigma_y^2}\right)}$$

is used, where where A is the amplitude, x_0,y_0 is the center and σ_x, σ_y is the spread in x and y, that is, how far the value will spread out from the point of the amplitude. This function gives good results, due to the way values are distributed, as seen in Figure 4.1 where $A = 1$, $x_0 = 0$, $y_0 = 0$, and $\sigma_x = \sigma_y = 1$.

In order to find the landing point, the Gaussian function is applied to every hoshimi point, OXY point and factory. After applying the function, the best starting point is simply found by choosing the point with the highest value.

Although this method is a significant improvement over the Barycenter approach described in Section 4.1.1, it too does not take terrain into consideration as well. This can make it perform badly in maps made to expose and exploit AIs using such algorithms. Therefore, a number of new methods has been investigated, which takes the shortcomings of the barycenter and gauss methods into account. These methods are described in the following sections.

4.2 Landing Point

This section contains details regarding selection of a landing point. First, the problem is formalized, followed by two different approaches to solving the problem. Both approaches are described and evaluated with regards

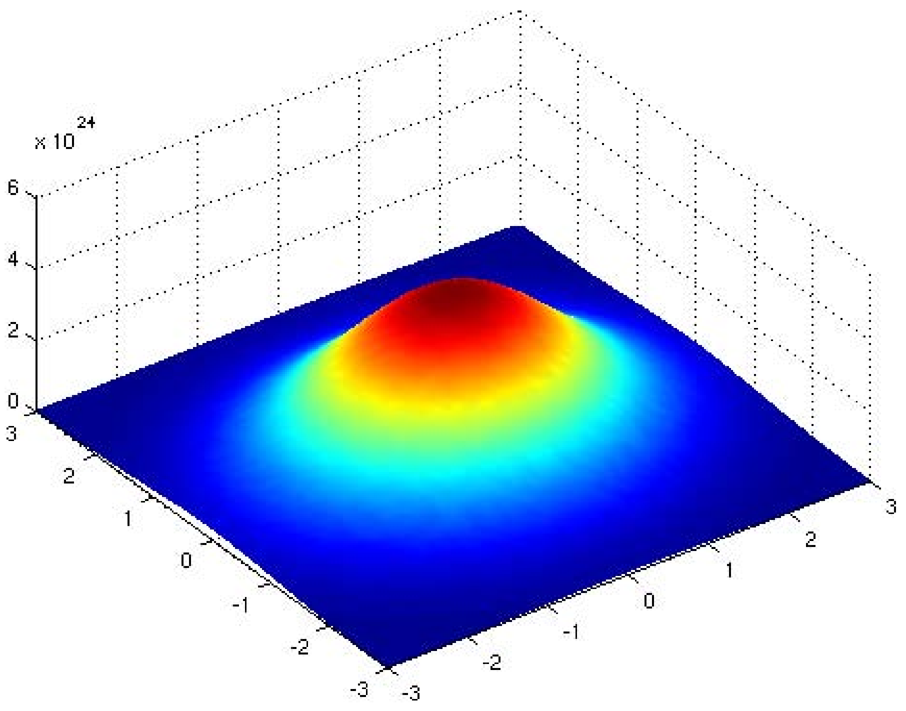

Figure 4.1: Two-dimensional gaussian function distribution [6].

to their use in Project Hoshimi. Lastly, the section is concluded with the selection of an approach for use in *Jetpack Attack*.

4.2.1 Formalizing the Landing Point Problem

Through examination of the problem, we have devised to following formalization: Given some map M with some number of objectives, find a starting point P from which the number of objectives reachable with less than or equal to D total distance walked is maximized. Another way to define the problem, is with the following maximization problem:

$$\operatorname*{argmax}_{P} ReachableObjectives(P, D, M)$$

where *ReachableObjectives* gives the number of objectives reachable in map M with less than or equal to D distance. No matter what point of found as the best starting point, it will always be on the location of an objective, e.g. an OXY or hoshimi point.

To put this is a more real-life usable form: Consider a city with several landmarks and a car with enough gas to drive some set number of miles, the problem is: If you could pick any spot in the city, where would you start to see the highest possible number of landmarks before you run out of gas? This translates to finding the spot where the AI bot can go to the highest number of objectives, within some set number of turns.

The following sections describe two different approaches to solving this problem.

4.2.2 Influence Mapping

This section describes influence mapping and how it can be used to decide where to place the landing point. The details of influence mapping is described along with strategies for applying it to our problem. Finally, the benefits and shortcomings of influence mapping is described.

An influence map is a spatial representation of the value of different areas of a map, that can be used to find an approximative result to the landing point problem. Influence mapping works by assigning influence to points in the map, and propagating the influence across the map, according to some falloff factor, to produce an overall picture of the value of the different areas in the map. The falloff factor decides how large a percentage of the influence of a given cell is propagated to adjacent cells. Using the influence map, it is possible to find points of high value, points of conflict, or areas of high value. Figure 4.2 is a visualization of an influence map. The map consist of two different sources of influence, represented by black dots and orange influence, and white dots and blue influence.

Given the nature of the problem of finding a landing point, influence mapping seems a perfect fit. By assigning values to different objectives, the location of highest influence can be found, which in turn is the most valuable starting location. The problem is in deciding what amount of influence for the different objects, and the algorithm for propagating the influence.

Through testing and experimentation, it turns out that influence mapping is not as good a solution to the landing point problem as first thought. By using a testing application where the influence and propagation can be modified at runtime, we found two major problems with the use of influence mapping in our context.

Figure 4.3 shows the two problems with using influence mapping for finding the landing point. In each side of the figure, the black cross denotes

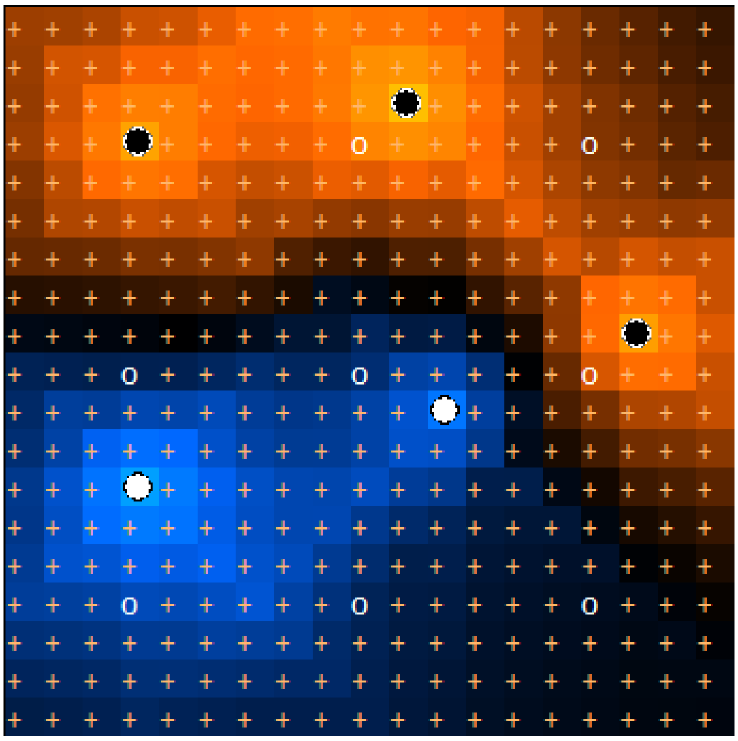

Figure 4.2: Visualization of an influence map. [7]

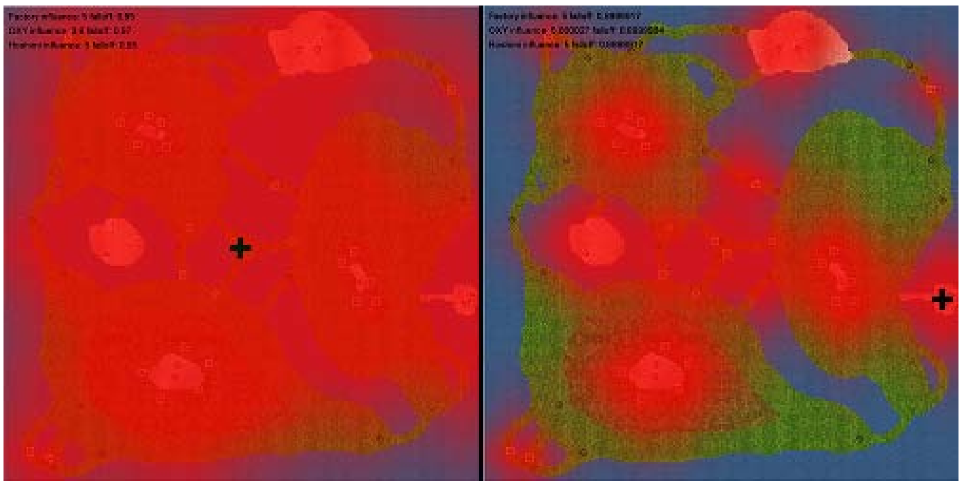

Figure 4.3: Two problems with influence mapping.

the point of highest influence.

The left side of Figure 4.3 shows the problem of having a too high falloff factor, that is, a falloff factor close to one. When a high falloff factor is used, it means that influence is propagated far across the map. When influence is propagated with almost no drop is values, the end result is almost the same as using weighted average Euler distances to the objectives. Because the *AI* bot is the only bot that can build *Needle* bots, it is more desirable to have a landing point close to a set of hoshimi points than it is to be close be in the closest distance to every objective.

The right side of Figure 4.3 shows the problem of having too low falloff values. With low falloff values, influence is not propagated very far across the map, which results in several areas of influence instead of one big area. With several areas of influence, one might think that it would be possible to achieve the desired goal. However, because influence is not spread very far, objectives that are very close together will produce very high influence values, and would therefore be chosen as a landing point. This is also not desirable, as it is more valuable to have five hoshimi points with relatively short distance, that three right next to each other.

Another problem, not depicted in the figure, is the OXY points. Because OXY points have unlimited amounts of OXY gas, it is never required to have access to more than one OXY points. Using a single influence map, this cannot be taken into account, which means that giving OXY points high influence would cause two OXY points next to each other to be regarded as a good starting position, which is not the case.

It might be possible to develop some way of working around these problems, e.g. by using several influence maps. However, because these problems stem from the way influence mapping works, the problems have lead us to believe that another approach would be better suited to solve the problem of finding the optimal landing point location.

4.2.3 Graph Approach

This sections describes how the problem of finding a landing point can be turned into a graph problem, and then solved.

The problem in 4.2.1 can easily be converted to a graph problem, by converting the map M to a graph, and using graph theory to evaluate the *ReachableObjectives* function. However, to create a graph that mimics the problem at hand, several factors has to be taken into account. These factors are:

- The type of graph to build: directed, non-directed, cyclic or acyclic.
- What are nodes in the graph: every objective point, hoshimi points, OXY points, etc.
- How are nodes connected in the graph: weighted or non-weighted.

The goal is to create a graph that resembles the actual problem. To do that, the graph needs to be cyclic, as there are no restrictions on that in the game. It also needs to be directed, as the length of a path in one direction can be different from the length of the same path in the opposite direction, due to airstreams. In order to make the edges correspond to the path in the map, they have to be weighted, with the weight being the length of the path. The graph must have a path for every walkable path in the game, which means it will most likely be a fully connected graph, unless the map consists of unreachable locations. As the graph will be used to find the optimal landing point, the graph will have a vertices corresponding to one type of objective, depending on which objective is most valuable.

Building the graph then, consists of deciding what objective to use as vertices, and then fully connecting the graph, and calculating the distance of every edge in the graph, based on the path in the map. This will most likely result in large number of paths having to be calculated. As an example, the map used in *Small Competition 2* has 30 hoshimi points. If building the graph with 30 hoshimi points as vertices, the number of paths having to be found is:

$$NumberOfPaths = 2 \cdot \sum_{i=1}^{N-1} i$$

that is, if N = 30

$$NumberOfPaths = 870.$$

Because the landing point has to be found as fast as possible, this can prove to be a problem if calculating the path takes too long.

Given a graph as described above we have devised two graph approaches to solving the problem. One is a brute force exhaustive search of every available subgraph, and the other is based on pruning the graph until a solution is found, rather than checking every subgraph. Each solution will be described in detail below.

4.2.3.1 Brute Force

An exact result to the maximization problem can be found by performing an exhaustive search of for the best subgraph in the graph; best meaning the subgraph with the highest number of vertices, that can be visited in less than the distance D. Finding the shortest distance needed to visit all nodes in the subgraph is the well-known Traveling Salesman Problem (TSP), which is a known NP-hard problem. However, both exact and approximative solutions to the TSP are available, and they all perform well on low vertex counts, so solving the TSP in our case will not be the most expensive part of finding the landing point. Something much more problematic is the sheer number of subgraphs, for which the TSP has to be solved. Consider a graph of n vertices; the number of unique subgraphs without repetitions of size k is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

This means that the number of unique subgraphs of size 1 to n can be found using

$$\sum_{k=1}^{n} \binom{n}{k}$$

which means that in a graph with 30 vertices, the number of unique subgraphs is

$$\sum_{k=1}^{30} \binom{30}{k} = 1073741823.$$

The incredibly high number of subgraphs that has to be searched in order to find the optimal subgraph is the breaking point for this approach. By searching subgraphs starting with the biggest down to the lowest, the optimal subgraph might be found quickly in some cases, but a worst case of $2^N - 1$ subgraphs to check makes this approach unusable for our needs.

4.2.3.2 Pruning

This approach relies on pruning of the edges of the graph in order to produce a result quickly. Although not exact solution of the maximization problem, the result is found significantly faster than using a brute force approach, which is of higher importance in our case.

The major problem with a brute force solution, is the high number of subgraphs that has to be examined to find the best solution. By continuously removing edges from the graph, the graph will eventually be split up into several different strongly connected components, that can be examined for their potential of being a candidate.

In order to understand this approach, it is important to understand the concept of strongly connected components. In graph theory, a directed graph with a path from each vertex to every other vertex is called a strongly connected graph. The strongly connected components of a directed graph are the strongly connected subgraphs of a graph. As an example, see Figure 4.4 where the two strongly connected components of the graph are marked by a blue circle.

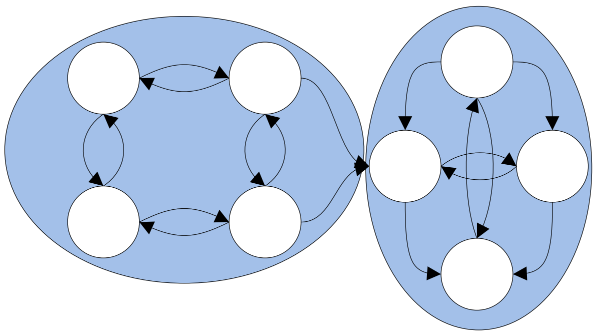
Figure 4.4: Example of strongly connected components.

When the strongly connected components have been found, the TSP path distance is calculated, and compared to the maximum allowed distance. If the TSP distance is less than the maximum distance, it use stored as a possible candidate for a starting point and removed from the graph, if not, it is left as part of the graph. By continuing in this manner, the graph will eventually be empty, and the best candidate can be found by selecting the candidate with the highest number of vertices, and picking the location of any vertex as the starting point. The following is an outline of the algorithm.

1. Given a graph as described in section 4.2.3.
2. Prune some number of edges, starting with the highest weighted edges.
3. Find every connected component in the graph pruned.
4. Solve TSP for every strongly connected component found; if the distance of the path is below the desired distance, save the component in

a list of candidates and remove it from the graph.

5. If the graph still contains vertices, go back to 2, otherwise, continue to 6.

6. Find and return candidate with the highest number of vertices.

Pruning the edges is done in a greedy manner, from the highest weight to the lowest. By always removing the edge with the highest weight, we hope to reach a global optimum by finding the largest possible strongly connected component within some maximum distance. However, the approach is not guaranteed to return the global optimum. When pruning the nodes, different approaches can be taken. One can simply choose to prune the highest weighted edge at every iteration, or to choose some set number of edges. Another way could be to prune a percentage of edges to achieve a faster albeit more inaccurate result. No matter what approach is taken, the algorithm is guaranteed to terminate, as a graph with no edges will have as many strongly connected components and the number of vertices.

Although this approach does not produce the exact result, finding the landing point using this approach can be done significantly faster than using a brute force method. Given a graph with V vertices and E edges, and that 1 edge is pruned every iteration of the algorithm, the algorithm will at most run E iterations. At each iteration, the strongly connected components have to be found. However, this can be done in $O(|V| + |E|)$ using Tarjan's algorithm [12] and is therefore not a problem.

As mentioned earlier, this approach is not guaranteed to produce a global optimum. For an example of a problem, consider the graph in Figure 4.5 and a maximum total distance of 20. Initially, the strongly connected component of the graph is the entire graph, which is not a candidate, as the TSP path distance is more than 20.

By using the prune algorithm on this graph, the highest weighted edge, i.e. the edge with weight 10, will be removed, and the strongly connected components of the graph is as shown in Figure 4.6. In this case, the largest strongly connected component that has a total TSP distance less than 20, contains 4 vertices.

However, neither the strongly connected components in Figure 4.6 are optimal. By removing a different edge, as shown in Figure 4.7, the largest

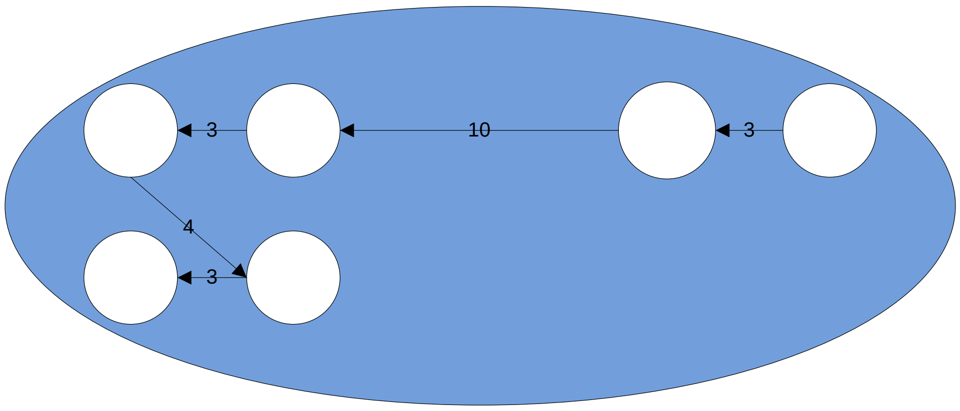

Figure 4.5: Example graph for the prune algorithms problem.

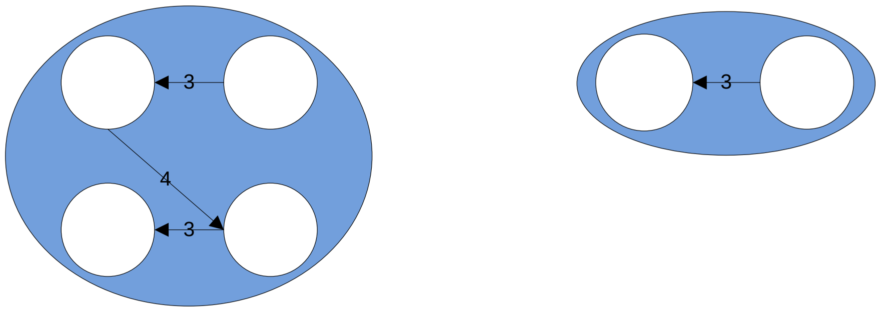

Figure 4.6: Faulty result found by prune algorithm.

strongly connected component that satisfies the TSP distance contains 5 nodes.

The prune approach falls short with graphs of this type, and is the main cause of why the prune approach produces results that are not exact. However, as mentioned earlier, exactness is less important than performance in our case. In this regard, the prune approach is a far more suitable approach to finding the landing point.

The problems with a brute force graph approach and an influence mapping approach has led us to believe that a prune approach would be best for finding the landing point in Project Hoshimi. Therefore, this will be the approach used to finding the landing point in *Jetpack Attack*. How the prune method has been implemented in *Jetpack Attack* is described in Chapter 5, and how well it performs is tested in Chapter 6

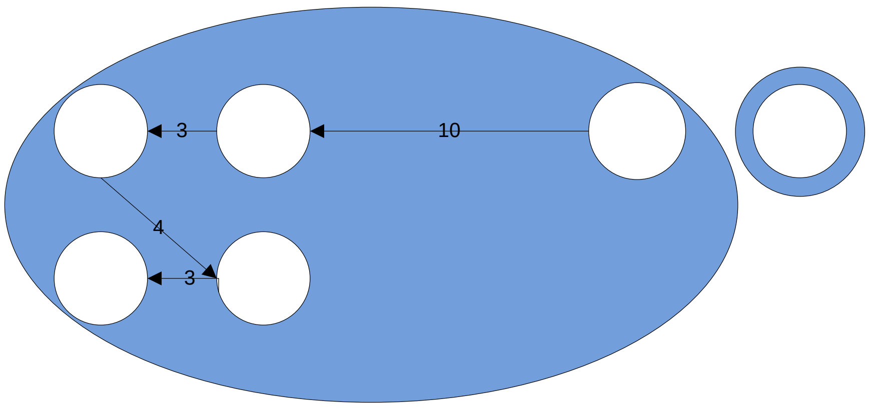

Figure 4.7: Correct result.

4.3 Mission Analysis

This section describes the techniques used to analyze the mission given at the beginning of a game. To find out what to focus on in the beginning of a mission, it is necessary to analyze the objectives, to be able to prioritize them, and discover what gives the most points in the game. This means that the objectives must be evaluated, and the most important information must be drawn out of them, to find out which ones are the most important for winning the game. To do this, it is necessary to structure the game in a manner such that it is possible to derive a focus point for a given mission. Therefore, a number of overall focus points have been found, which are described in the following section. These focus points are the formalized in to a model which can be used in the actual implementation of *Jetpack Attack*. This model is described in Section 4.3.2. The outcome of the analysis of the mission, is a prioritized list of the focus points, to use for deciding which objectives to solve first.

4.3.1 Focus Points

As described in Section 2.2.1, there are four active objectives in Project Hoshimi, which can be translated directly in to focus points, for the analysis of the mission. Therefore, the focus points chosen are factories, hoshimi, navigation, and finally unique navigation, which covers all the active objectives. Each of these focus points are made up of different factors, which each influence the importance of the overall goal. These factors are taken into account when doing the analysis, to gain an idea of what to choose as the main focus point.

One common factor the focus points share, is the score of the given objective, i.e. how many points can be made by completing it. It is important to note that this deciding factor is the same unit for all focus points, meaning for example that a score of less than 1000 points in the factory focus point, is classified as the same in the hoshimi focus point. This is done to be able to compare the different focus points.

4.3.1.1 Factories

The factories focus point deals with achieving the *PlayerFactoryDestruction* objectives. As described previously, these objectives state that the player must destroy more than half the factories in the map. To determine whether or not factories are important for winning the game, there are four factors to consider.

- **Score:** The score given for solving the objectives. The higher the score is, the higher priority factories should get.
- **Number of Bots:** The number of bots needed to solve the objective. The higher number of bots necessary, the higher priority, since the more bots needed, the sooner they should be built and deployed.
- **Kill Percentage:** How big a percentage of the total number of factories that must be killed. If this number is low, it means less effort is necessary to solve the objective, and it should therefore be prioritized lower.

Each of these factors contribute in prioritizing the factories goal, and they are all measures for how much effort, solving this objective would take. For example, if the number of bots needed is high but the score is categorized as low, it signifies that the factories goal should not be prioritized high, because it would take a lot of effort solving it, with little reward.

4.3.1.2 Hoshimi

The hoshimi focus point is about gathering OXY gas and transferring it to hoshimi points. The following factor influences the importance of hoshimi points.

- **Score:** The total score obtainable by filling the hoshimi points with OXY gas. The higher the score is, the higher the priority the hoshimi focus point should get.

The reason for only choosing score as a deciding factor for the hoshimi goal, and not the number of hoshimi points in the map, is that these two numbers are directly proportional, since each filled hoshimi point grants the same number of points, and also to be able to compare this goal to the other goals, which also uses score as a measure, score has been chosen over the number of hoshimi points.

4.3.1.3 Navigation

This goal deals with solving navigation objectives. To prioritize this goal, the factors *Number of Bots* and *Score* has been chosen.

- **Number of bots:** The number of bots needed for solving the objectives. Once again, the lower the number of bots, the higher priority.
- **Score:** The total score of the navigation objectives.

Like the factories objective, if the number of bots needed is high, and the score is low, this goal will not be prioritized, since it would take too much effort to solve, compared to other goals.

4.3.1.4 Unique Navigation

This goal is very similar to the navigation goal, but instead deals with unique navigation objectives. The following factors have been deemed important for this goal.

- **Distance:** The distance to be traveled to solve the unique navigation objectives. If the distance is low, it takes less effort to complete the objective should be prioritized higher.
- **Score:** The total score for solving the unique navigation objectives.

Again, these factors signify the effort of solving the goal versus the reward for doing so.

4.3.2 Model

This section describes the formal model used for analyzing the mission given at the beginning of a game. For this analysis, it has been chosen to use *Bayesian Networks*, and the chosen structure, is the Hierarchical Naive Bayes classifier, inspired by [8]. This structure is a modification of the Naive Bayes classifier, which is used to classify instances of a given problem, from a set of attributes, describing the problem. It usually consists of a class node, which is the root of the network, representing the variable to be classified, and a set of leaf nodes which are the attributes. In the hierarchical version, a set of intermediate nodes are introduced between the root and the leaf nodes.

The classifier for *Jetpack Attack* is depicted in Figure 4.8.

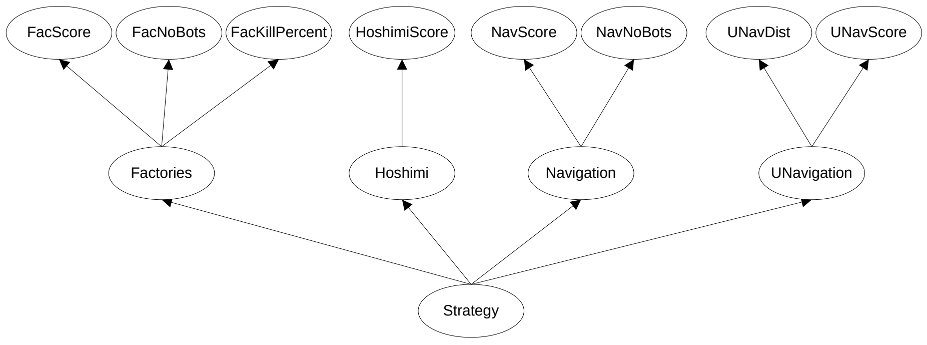

Figure 4.8: Hierarchical Naive Bayes Classifier.

In this network, the class node, *Strategy*, is the one being classified, and it consists of four states, i.e. the four focus points from Section 4.3.1, and the attributes are the subgoals of each goal. The structure of the network allows for inference, based on the evidence received from the mission in the beginning of a game.

The main challenge in using this network, is choosing the probabilities for each of the Conditional Probability Tables (CPT) of the nodes. Normally these would be built by learning from data by playing against other players. However, as mentioned previously, since this is a competition, other players are not willing to share their strategies. Therefore, we have to estimate the probabilities ourselves, and an example of this can be seen in Table 4.1, which shows the CPT for the *FacKillPercent* node.

As the table shows, the *FacKillPercent* node has three states, i.e. low,

Factories	Low	Medium	High
Low	0.7	0.1	0.05
Medium	0.25	0.8	0.25
High	0.05	0.1	0.7

Table 4.1: FacKillPercent CPT

medium and high. This is an effort to discretise the node, and this in itself can also be hard, since determining what e.g. a *High* kill percentage means is not trivial. The table also shows that the *Factories* intermediate node has the same states as the *FacKillPercent* node. In this case this symbolizes the probability of recommending the factory goal. The rest of the CPTs for the attributes are structured similarly to that of the *FacKillPercent* node, and can be seen in Appendix C. The actual values of *Low*, *Medium* and *High* can be seen in Appendix D. These values are derived from examining six different, official maps, and then estimating the values, based on the maps.

To use the network, one simply has to insert the data from the mission, and the run inference on the network. How this network performs in practice is described in the following section.

4.3.3 Experiments

To test how well the Bayesian network described in the previous section performs in practice, a number of tests have been conducted. These tests are described in detail in Chapter 6 but as the results are important for the remainder of the report, they will briefly be discussed here. The tests were structured to test what focus point would be chosen in different maps. To do this, data from each map was inserted as evidence in to the network, inference was run, and the resulting focus point was noted. For every map we encountered, i.e. all the official maps, the focus always turned out to be hoshimi points. The reason for this is, that in every map there was always a large number of hoshimi points, and therefore a lot of points to gain from this objective. Since we are using the same unit for comparing the *Score* of each focus point, and the score of hoshimi points always has a higher score than the others, it was selected as the best focus point every time. It also makes sense to always focus on hoshimi objectives, since they always provide a way of scoring a large number of points.

The initial analysis of the mission will always end up with the same result, and is thus rather pointless. Therefore, the Bayesian network will not be implemented in *Jetpack Attack*, as it would bring a completely unnecessary overhead to the implementation, which is needed to be computationally light, because of the time-based structure of the game.

4.4 Bot Behavior

This section describes how the behavior of the various bots can be modeled and controlled using Finite State Machines (FSM). The information about FSMs is based on [10].

A common and simple way to model behavior, is using FSMs. An FSM is a collection of states and transitions between states. A very simple FSM model of a door can be seen in Figure 4.9.

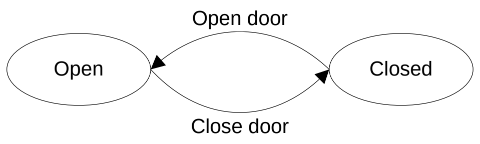

Figure 4.9: A Simple FSM modeling a door.

FSMs can similarly be used to model the behavior of a bot, by splitting the behavior into different states, and defining transitions from state to state. Consider a bot that has the following actions available: Receive Order, Attack, Flee, Move, and Collect. By modeling the bot using FSMs, implementing the bot can be done in a very elegant and almost trivial manner. An example FSM that can be used to model the behavior of such a bot can be seen in Figure 4.10.

The bot in Figure 4.10 immediately starts waiting for orders after it is created, by switching to the *WaitForOrders* state. When an order is received, the order is prepared in some fashion, and the bot switches to the *Move*, and moves to whatever location it needs to be at. When it is at the desired position, the state is changed to either *Collect* or *Attack*, depending on what the objective is. When the bot is done collecting or attacking, the state is switch back to *WaitForOrders* and so on. At any point, if the bot sees an enemy, it enters the *Flee* state and runs away. When the enemy is no longer seen, it returns to the state it left to enter the flee state. Implementing this behavior can be done using a simple switch case, as shown in Listing

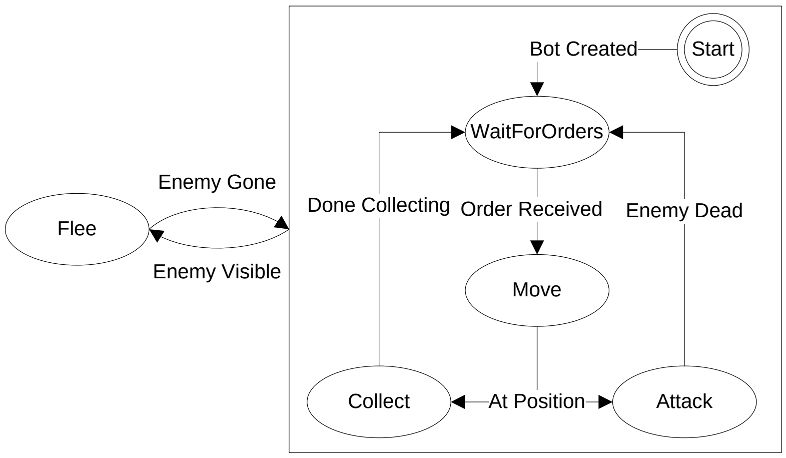

Figure 4.10: An example of a FSM modeling the behavior of a bot.

4.1.

```
enum BotStates
{
    WaitingForOrders,
    Move,
    Flee,
    Attack,
    Collect
}

void BotLoop()
{
    BotStates state = BotStates.WaitingForOrders;

    while (true)
    {
        switch (state)
        {
            case BotStates.WaitingForOrders:
                // Wait for orders logic here, when an order is received,
                // change to BotStates.Move.
                break;
            case BotStates.Move:
                // Move the bot, when at the desired position, change to
                // either BotStates.Attack or BotStates.Collect, depending
                // on the type of order.
                break;
            case BotStates.Flee:
                // Run away from the enemy. When the enemy is no longer
                // visible, change back to the previous state.
                break;
            case BotStates.Attack:
                // Attack the enemy. When the enemy is dead, change to
                // BotStates.WaitingForOrders.
                break;
            case BotStates.Collect:
                // Collect resources. When done collecting, change to
                // BotStates.WaitingforOrders.
                break;
            default:
                break;
        }
    }
}
```

Listing 4.1: Sample FSM Implementation

The code in Listing 4.1 shows part of the bot implementation in C#. In this case, the different states of the bot are switched over using an enumeration of the different states. In each different case of the switch, the corresponding logic is written, along with any state changes. This general method of implementing FSMs is easy to implement and understand, and will be used to implement all bots used in *Jetpack Attack*.

This concludes the description of some of the various techniques used for the AI in *Jetpack Attack*. The next chapter describes how these techniques have been implemented.

Chapter 5

Implementation

This chapter describes the implementation of some of the key features of *Jetpack Attack*. It is not meant to be an exhaustive review of the entire implementation, but rather a brief overview, highlighting interesting elements of *Jetpack Attack*. This includes how the *Landingpoint Finder* and the central control unit, the *MissionController*, has been implemented, and finally an example of a bot-implementation.

5.1 Design

Before discussing the actual implementation of *Jetpack Attack*, it is necessary to describe the basic structure of the implementation. This section describes this initial design of the implementation, along with a brief description of the Project Hoshimi SDK.

5.1.1 Project Hoshimi SDK

To implement a strategy in the Project Hoshimi SDK, it is necessary to create a class which inherits from *Player*, and create event handlers for the two events *ChooseLandingPoint* and *WhatToDoNext*. The first one is fired once in the beginning of a game, and when this happens, the event handler must choose where the landing point must be in the map. The second event is fired every four turns throughout the game, and is where the actual strategy is implemented. There are two central pieces of information available from the beginning of the game, which are the objects *Mission* and *Terrain*. The first one is, as the name states, the mission for the given game, which holds all information about the objectives, such as type, location and score. The second is the map used in the mission, and has information about the speed modifiers for all points in the map, and the location of hoshimi and OXY points.

5.1.1.1 Bots

The bots described in Section 2.1, are all generic types of bots. When developing an AI for Project Hoshimi, it is necessary to customize the bots, by altering the characteristics from the following list.

- **Container Capacity:** the maximum number of OXY gas units the bot can carry.
- **Collect/Transfer Speed:** how fast the bot can collect OXY gas and transfer it.
- **Scan Range:** how far the bot can see on the map.
- **Maximum Damage:** how much damage the bot can deal to opponents.
- **Defense Distance:** how far away from itself the bot is able to do damage
- **Constitution:** the health of the bot. If this reaches zero, the bot is destroyed.
- **Shield:** when bots have a positive shield value, it and the friendly bots are unaffected by attacks. However, the shield value decreases with each attack.

The bots which are able to put points into *Defense Distance* and *Maximum Damage* are the ones used for attacking and destroying factories. They can also attack the bots of the opponent, and are therefore very useful for hindering him in completing objectives.

Each type of generic bot, has a predefined template that they must adhere to, which describes maximum values for the different characteristics and a number of possible actions they can perform. As an example, the template for a *Collector* bot, can be seen in Table 5.1.

Characteristic	**Value**	**Actions**
ContainerCapacity	20	Self Destruct
CollectTransferSpeed	5	Move
Scan	5	Stop Moving
MaxDamage	5	Defend
DefenseDistance	12	Collect OXY
Constitution	50	Transfer OXY
Total	50	

Table 5.1: Collector Template

The numbers is in the *Value* column, are the maximum number the given characteristic can have for the collector. The *Total* row signifies the total amount of points that can be put into the characteristics for the bot, so it is necessary to decide which characteristics are most important, since not all of them can be awarded the maximum number of points. Also notice that the shield characteristic is left out. This is because only the AI bot and the Needle bot can have shields. The *Actions* column, states what the bot is able to do. For the *Collector* bot this is self destruction, moving, defending itself and loading and unloading OXY gas. For a complete list of templates for all bots, see Appendix B. Now that the Project Hoshimi SDK has been described, the actual implementation of *Jetpack Attack* can be presented. First, the basic structure is described, and then parts of the implementation is shown in greater detail.

5.1.2 Jetpack Attack

This sections describes the overall structure and design of the implementation of *Jetpack Attack*.

5.1.2.1 Controller

The main class of *Jetpack Attack* is the *Controller* which inherits from *Player*, and as mentioned previously, has handlers for the two events *Choose-LandingPoint* and *WhatToDoNext*. The first is responsible for setting the property *LandingPointWanted*, which is were the landing point for the game will be chosen. The implementation of the *Landingpoint Finder*, which makes this choice, is described in Section 5.2. The second event, as mentioned, is the main loop of the game, which is fired every 4 turns, and where all actions performed by the bots are initiated. The *Controller* class also controls the *AI* bot, which is responsible for building the other bots. The *Controller* has three basic states it can be in, i.e. *Building*, *Defending* and *Hoshimi*. The *Building* state is where the *AI* bot builds other bots, and what to build is dictated by the class *MissionController*, described in Section 5.1.2.2. The *Controller* goes in to the *Defending* state, when an enemy is nearby, and in this state the *DefenseNeedle*, described in Section 3.2.1, is built and the *AI* bot stands still, until the enemy is disposed of. Finally, in the *Hoshimi* state, the *AI* bot moves between the hoshimi points in the map, and builds *InjectionNeedles* used to load OXY gas in to the ground. The *Controller* can switch between these states, depending on what is necessary at a given point in the game, and what information is received from the *MissionController*, described in the following section.

5.1.2.2 MissionController

As described in Section 3.2, a central class is needed to keep track of the status of the mission. This class, called the *MissionController*, is responsible for generating and managing the different tasks for a given mission. When the game starts, the *MissionController* is given the *Mission* object, and starts to analyze the mission. This is done by first creating a list of tasks for each of the bots used for solving objectives, i.e. the *Destroyer*, the *Scout* and the *Transporter*, and then populating these lists with information from the given mission. Every time an objective is found in the mission, it is broken in to separate tasks, one for each location that needs to be visited or, in the case of factories, destroyed. Each task is given a priority, based on how important they are for scoring points and doing well with the given mission. This prioritization can be very hard, since there are a lot of factors which come into play, when deciding which objectives are important for the mission. When the tasks have been prioritized, they are analyzed, and compared to the already existing tasks, to see if the target of the original task is close to the targets of any other task. If so, the target from the original task, is added to the found task. Otherwise a new task is added. If the objective requires a specific bot, this is set as the bot type for the task, and if this is not so, a bot will be selected. As described previously, for unique navigation tasks, the bot chosen will be a *Scout*, if it is a non-unique navigation objective or a factory objective, the *Destroyer* will be used.

Apart from the initial analysis of the mission, the *MissionController* is also responsible for delegating tasks to the different bots in the game. When a bot needs a new task, it will simply ask the *MissionController* for the next task for that specific bot type, and it will look through the list of task for that bot, and return the one with highest priority, which has not been completed or picked up by another bot. If no such task exists, meaning all tasks are completed, or at least picked up by another, the bot will selfdestruct.

The *MissionController* is implemented as a *Singleton* class, to enable access of the same instance of it from all parts of the game.

5.1.2.3 Bots

Apart from inheriting from the generic bot they stem from, each bot in *Jetpack Attack* also implements the interface *IBot*. This interface only has one method signature, i.e. the *DoNext(Controller player)* method, which is simply the main method of each bot, and this is called every time *WhatToDoNext* is fired. The reason for passing the *Controller* object to the method, is to be able to access data about the game, e.g. the list of the enemy's bots which are currently visible. All bots used in *Jetpack Attack* are implemented as FSMs, which are described in Section 4.4. The following describes the different bots.

Scouts

The *Scout* bot is a derivative of the *Explorer*, and is a very fast bot, which ignore movement-impairing parts of the terrain. Therefore, as mentioned above, it is used for unique navigation objectives, since these objectives require the bot to travel a long distance, within the game's time limit. *Scouts* are not able to defend themselves, so therefore they can call for help.

Destroyers

The *Destroyer* bot is a derivative of the *Collector*, and is a very versatile bot, and is therefore both used for navigation and factory objectives. The reason they are viable for navigation is that these objectives do not require one specific bot to travel to all targets, and therefore a *Collector* can be built for each target, of each objective. Because of their versatility, *Collectors* are the most used bots in *Jetpack Attack*, and therefore they are able to help each other finish tasks. When they finish their own tasks, and no new ones are available, instead of immediately selfdestructing, they can ask the *MissionController* if there are any tasks close to it. If that is the case, they can start going to the targets of that task.

Transporters

The *Container* derivative *Transporter* is very essential for scoring points in the game, since its main responsibility is to transport OXY gas from OXY points, to hoshimi points, and then unloading it. Since it is also possible that navigation objectives can require a *Container* to be completed, the *Transporters* are also able to do navigation tasks.

5.1.2.4 Map

The map of any given game in Project Hoshimi has a fixed size of 200x200 pixels. Each pixel has a terrain type which can be *Normal* which takes two turns to pass, *Slow* which takes 3 turns to pass, *Very Slow* which takes 4 turns to pass, and finally *Water* which can not be passed. To enable pathfinding in the map, the class *MapGrid* is used. It consists mainly of a 200x200 two-dimensional array, which holds *MapNodes*, which contain information about the cost of moving across a given point in the map, and other relevant data, such as adjacent and parent nodes. For pathfinding, the A* algorithm[5] is used, and is implemented in the *Pathfinder* class. Because of the size of the map, it is not viable to use A* for calculating all paths in the game, as it would take too much time. Therefore it is only used to calculate the path for the *AI* bot and the *Bodyguard*. For other bots, the pathfinder built into the SDK is used. The *MapGrid* is, like the *MissionController*, implemented using the *Singleton* design pattern.

This concludes the design portion of this chapter. The following sections describe the finer details of some the important features of *Jetpack Attack*.

5.2 Landing Point Finder

This sections describes the implementation details for the landing point finder, which is responsible for finding the landing point.

The landing point finder consists of three classes: *Graph*, *LPGraph* and

LPFinder. The *Graph* class is a generic adjacency-list based graph implementation, used as a base class for the *LPGraph* class. The *LPGraph* class is responsible for handling the graph functionality specific for finding a landing point, such as pruning the graph, and finding the strongly connected components. The final class is the *LPFinder* class, which is the actual class responsible for finding the landing point. This class uses an instance of the *LPGraph* class. The class diagram of the landing point finder can be seen in Figure 5.1.

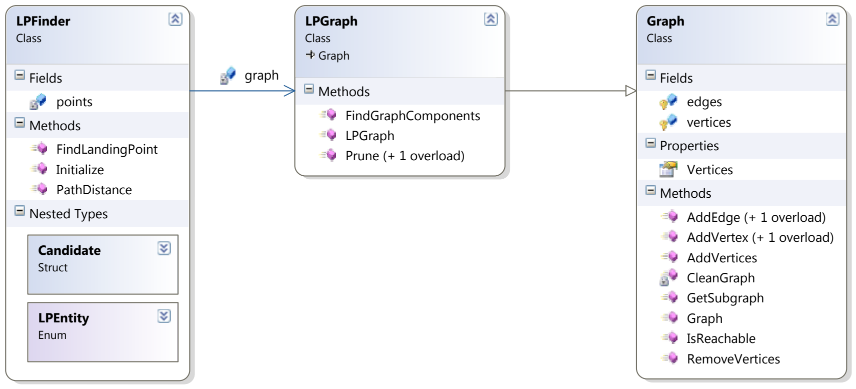

Figure 5.1: Class diagram of the landing point finder.

Both the *Graph* and *LPGraph* are simple and uninteresting, and will therefore not be described. However, during implementation of the *LPFinder* class, a few changes had to be made to improve performance during initialization. To shed some light on these changes, the rest of this section will focus on the details of the *LPFinder* class.

In order to find the landing point using the prune method described in Section 4.2.3.2, a graph has to be built first, and then pruned until empty. Building the graph is done in the *Initialize* method, which can be seen in Listing 5.1.

```
public void Initialize(Point[] points, int pruneDistance)
{
    this.points = points;
    graph = new LPGraph();
    // Add the vertices to the graph
    foreach (Point point in points)
    {
        graph.AddVertex(point);
    }

    // Add the edges
    for (int i = 0; i < graph.Vertices.Count; i++)
    {
        Vertex currentVertex = graph.Vertices[i];
        for (int j = i + 1; j < graph.Vertices.Count; j++)
        {
            int manDistance = Utilities.ManhattanDistance(currentVertex. ↩
                location, graph.Vertices[j].location);
            if (manDistance <= pruneDistance)
            {
                graph.AddEdge(currentVertex, graph.Vertices[j], PathFinder. ↩
                    GetDistance(currentVertex.location, graph.Vertices[j]. ↩
                    location));
            }
            else
            {
                graph.AddEdge(currentVertex, graph.Vertices[j], manDistance ↩
                    * 5);
            }
        }
    }
}
```

Listing 5.1: LPFinder.Initialize Implementation

Initialization of the *LPFinder* class is simple. Firstly, the vertices are added by using a list of points that is passed as an argument to the *Initialize* method. After the vertices have been added, the edges are added to the graph. However, some changes had to be made to the calculation of distances, due to performance issues. As mentioned in Section 4.2.3, the original goal was to have the weight of edges be the actual distance required to travel vertex to vertex. However, during implementation, we quickly found out that this simply would not be possible without spending most of the game time on building the graph. Therefore, to speed the graph building up, we decided that some distances would have to be estimated. Specifically, as the longer distances will get pruned quickly, and they require more time to calculate, they do not need to be exact. The decision of whether a distance has to be exact or not is based on the Manhattan distance[1] of the edge. If the Manhattan distance is higher than the *pruneDistance* argument passed to the *Initialize* method, the Manhattan distance multiplied by five is used, instead of the actual distance. On the other hand, if the distance is lower, the actual distance is calculated using the *Pathfinder* class. This can be seen in Lines 17-25. After *Initialize* has been run, the graph is built and the landing point can be found by calling *FindLandingPoint*.

The *FindLandingPoint* method is the implementation of the actual algorithm for finding the landing point. As described in Section 4.2.3.2, the algorithm works by pruning the graph, finding strongly connected components, and checking them for their viability as a candidate. The loop in *FindLandingPoint* where the candidates are found can be seen in Listing 5.2.

```
// snip

while (graph.Vertices.Count > 0)
{
    Graph[] components = lpgraph.FindGraphComponents();

    foreach (Graph component in components)
    {
        Vertex[] path = TSPSolver.Solve(component).ToArray();
        int distance = PathDistance(path);
        if (distance < maxWeights)
        {
            Candidate candidate = new Candidate();
            candidate.path = path;
            candidate.weight = distance;
            candidates.Add(candidate);
            graph.RemoveVertices(path);
        }
    }
    lpgraph.Prune(prunePercentage);
}

// snip
```

Listing 5.2: Part of LPFinder.FindLandingPoint Implementation

The code in Listing 5.2 is responsible for finding every possible candidate. It works by first finding every strongly connected component in the graph, and then checking each component to see if it satisfies the maximum distance requirement. If it does, the candidate is added to a list of candidates, and the vertices in candidate are removed from the graph. The graph is then pruned, and the loop is run again. This continues until no vertices are left in the graph. The best candidate is then found, and a starting vertex is selected from the candidate.

The speed versus exactness of the *LPFinder* can be controlled by the arguments *pruneDistance* and *prunePercentage* of the *Initialize* and *FindLandingPoint* method respectively. As mentioned earlier, the *pruneDistance* is used when deciding whether to calculate the actual weight of an edge, or using a Manhattan distance estimate. This means that a higher *pruneDistance* will result in more paths being calculated, which will result in a better result when pruning. The *prunePercentage* argument specifies how many edges are pruned at each iteration. A higher value removes more edges at each iteration, and therefore find more components faster. However, by being greedy when removing edges, it is possible that too many edges are removed, which results in a more inaccurate result as well.

Using this implementation, we are able to find our landing point with the first 30 turns, with *prunePercentage* set to 10% and *pruneDistance* set to 40. The experiments performed, to further evaluate the implementation of the *Landingpoint Finder*, can be seen in Chapter 6.

5.3 Mission Controller

This section describes the implementation of the *MissionController*, the class which keeps track of the progression of the mission for a given game. As described in Section 5.1.2.2, this class must also create the tasks for the other bots to solve, when the game is started. The implementation of these two features, are described below.

5.3.1 Initialization

The initialization of the *MissionController* deals with generating tasks for the bots to complete. As described previously, each of the three bots, *Scout*, *Destroyer* and *Transporter*, have their own list of tasks, which must be populated in the beginning of the game. This is done in the method *AnalyzeMission*, shown in Listing 5.3.

```
 1 //snip
 2 foreach (BaseObjective obj in mission.Objectives)
 3 {
 4     if (obj is NavigationObjective)
 5     {
 6         NavigationObjective navObj = obj as NavigationObjective;
 7         BotType type = BotType.Collector;
 8         if (navObj.BotType != BotType.Unknown)
 9         {
10             type = navObj.BotType;
11         }
12         foreach (Point p in obj.GetObjectiveLocations())
13         {
14             AddNavigationTask(Settings.NavPriority, p, type);
15         }
16     }
17     //Similar code for UniqueNavigation and PlayerFactoryDestruction ↩
           objectives
18  }
19 //snip
```

Listing 5.3: MissionController.AnalyzeMission Implementation

First all objectives are examined, and for each location in the objective, the method invokes the appropriate task generator, in the example this is the *AddNavigationTask* method. The task generators take a priority, a location and a bot type, and from this either generates a new task, or finds one close to the location, and adds the location to that task. As described earlier, the default bot for *Navigation* objectives, is the *Collector* derivative, *Destroyer*. Therefore, this is set in line 7, and only changed if the objective calls for something different. The tasks generating method, *AddNavigationTask* method is shown in Listing 5.4.

```
//snip
if (typeOfBot == BotType.Collector)
{
    Task tsk = new Task();
    foreach (Task task in DestroyerTasks)
    {
        if (task.IsUnique || task.Completed)
        {
                continue;
            }
            bool suitableTaskFound = true;
            foreach (Point p in task.Targets)
            {
                if (Utilities.ManhattanDistance(p, target) > 50)
                {
                    suitableTaskFound = false;
                }
            }
            if (suitableTaskFound)
            {
                tsk = task;
            }
        }
        if (tsk.ID != 0)
        {
            tsk.AddTarget(target);
            tsk.UpdatePriority(priority);
        }
        else
        {
            DestroyerTasks.Add(new Task(NewId(), priority, BotType.Collector, ↩
                target, false));
        }
}
//snip
```

Listing 5.4: MissionController.AddNavigationTask Implementation

This example only shows what happens if the bot type is *Collector* but it is similar for *Explorers* and *Transporters*. All *Destroyer* task are examined and if they are not *UniqueNavigation* objectives or complete, it is checked if all targets of the task is close to the new location. If so, the new location is added to the task and the priority is updated. If not all targets are close to the new location, a new task is generated and added to the list of *Destroyer* tasks.

5.3.2 Mission Progress

Once the mission has been analyzed, the main responsibility of the *Mission-Controller* is to keep track of the tasks, and inform the *Controller* of what to build next. In the beginning of each turn, the *Controller* invokes the *NextBot* method in the *MissionController* which returns the bot type of the next bot to be built. This method can be seen in Listing 5.5.

```
//snip
int curPriority = 0;
Task nextTask = new Task();
NextBotType result = NextBotType.None;
foreach (Task task in DestroyerTasks)
{
    if (!task.Completed && !task.Ongoing)
    {
        if (task.Priority > curPriority)
        {
            nextTask = task;
            curPriority = task.Priority;
        }
    }
}
//Similar code for Scout and Transporter tasks

if (nextTask.Priority == 0)
{
    int remainingBots = 40 − NoOfBots;
    int neededHunters = 5;
    if (NoOfTransporters < remainingBots)
    {
        result = NextBotType.Transporter;
    }
    else if (NoOfHunters < neededHunters)
    {
        result = NextBotType.Hunter;
    }
    else
    {
        result = NextBotType.None;
    }
}
else
{
    switch (nextTask.Bot)
    {
        case BotType.Collector:
            result = NextBotType.Destroyer;
            break;
        case BotType.Container:
            result = NextBotType.Transporter;
            break;
        case BotType.Explorer:
            result = NextBotType.Scout;
            break;
        default:
            break;
    }
}
return result;
```

Listing 5.5: MissionController.NextBot Implementation

The *NextBotType* type, is an enumeration containing the different bot types, which are available. The *NextBot* method first runs through all tasks, and checks if there are any which has not been picked up by a bot. If there are more of these, the one with the highest priority is chosen. If all tasks are taken, the method checks if either *Hunter* bots or *Transporters* are missing. If this is the case, the given bot is set as the result. Otherwise, the type *None* is returned, signifying that it is not necessary for the *Controller* to build anything the given turn.

5.3.3 Other Functionality

Apart from the two main features of the *MissionController* described above, it also has a number of minor functionalities worth mentioning. For instance, the *MissionController* is also responsible for keeping track of the hoshimi points, which ones does not have needles on them and which ones do not. It also has auxiliary methods, the bots can use, such as *NearestDestroyerTask* which takes a location, and returns the task which is closest to that location. This is helpful when a *Destroyer* is finished with its own task, and there are no new tasks, then the *Destroyer* can help with other tasks. Another functionality is the *ReportKill* method, for bots to report if they have killed an enemy which was not in their task. Then this enemy can be removed from a task, so time is not wasted going to that location.

5.4 Bot Example

This section describes the implementation of the *Transporter* bot as an example of how bots are implemented in *Jetpack Attack*.

The *Transporter* bot has been implemented according to the model described in Section 4.4. The *Transporter* has the following 6 states:

- GoingToOXY - The transporter is moving to an OXY point.
- GoingToNeedle - The transporter is moving to a needle.
- Collecting - The transporter is collecting OXY gas from an OXY point.
- Depositing - The transporter is depositing OXY gas to a needle.
- FollowingAI - The transporter is following to AI while waiting for an open needle.
- Tasking - The transporter is solving an objective.

These 6 states model all the needed functionality of the *Transporter*. As described in Section 4.4, the bot is implemented using a simple switch statement, which is implemented in the *DoNext* method that is called by the *Controller* every 4th turn. Part of the implementation of the *Transporter* can be seen in Listing 5.6.

```
[Characteristics(CollectTransfertSpeed = 5, Constitution = 15, ↩
    ContainerCapacity = 50)]
class Transporter : Container, IBot
{
    Random rand = new Random();
    Task currentTask = null;
    Point target = Point.Empty;
    public TransporterStates transporterState = TransporterStates. ↩
        GoingToOXY;

    public void DoNext(Controller player)
    {
        if (HitPoint < 8 && currentTask != null)
        {
            currentTask.Reset();
        }
        if (currentTask == null)
        {
            currentTask = MissionController.NextTransporterTask();
            if (currentTask != null)
            {
                currentTask.Ongoing = true;
                transporterState = TransporterStates.Tasking;
            }
        }

        switch (transporterState)
        {
            case TransporterStates.GoingToOXY:
                //snip
                break;
            case TransporterStates.GoingToNeedle:
                //snip
                break;
            case TransporterStates.Collecting:
                //snip
                break;
            case TransporterStates.Depositing:
                //snip
                break;
            case TransporterStates.FollowingAI:
                // snip
                break;
            case TransporterStates.Tasking:
                // snip
                break;
            default:
                break;
        }
    }
}
```

Listing 5.6: Transporter bot Implementation

When *DoNext* is called, the bot first checks to see if it is dying so that it can give up its task. This is done in Lines 10-13. After that, the bot checks for any available task. If a task is received, the bot changes to the *Tasking* state, and solves the objective, as seen in Lines 14-22. The bot then goes to whatever state it is in, and does whatever is needed. The actual implementation of each state has been omitted in this description, as it is trivial and uninteresting.

Every bot has been implemented using this method. The states of each bot varies, but the overall structure is shared among all bots. It should also be noted that the *Controller* also has been implemented as a bot, i.e. using FSMs to control the flow of the application.

This concludes the description of the most interesting elements of the implementation of *Jetpack Attack*. Experiments designed to evaluate parts of this implementation are described in the following chapter. An evaluation of the entire implementation can be found in Chapter 7.

Chapter 6

Experiments

This chapter describes different experiments performed on *Jetpack Attack*, to test various parts of the system. The Bayesian network used to analyze the mission, is tested to evaluate the how well it works on different maps, and the *Landingpoint Finder* is tested with regards to performance and score. Finally, the competition aspect of Project Hoshimi is discussed, along with how well *Jetpack Attack* performed against other players' AI.

6.1 Landingpoint Finder

This sections describes the experiments performed on the *Landingpoint Finder*, to evaluate how well it performs with regards to two criteria, i.e. performance and score.

6.1.1 Performance

To test the performance of the *Landingpoint Finder*, it has been tested on graphs of various sizes. This test will help show how the approach scales to problems with higher vertex count than used in Project Hoshimi.

The performance test has been run on randomly generated graphs with a vertex count ranging from 10 to 150 in increments of 10. Every edge in the graph has been randomly generated as well, with weights ranging from 5 to 200. In every test, the *Landingpoint Finder* has been run with 500 as the maximum weight. Furthermore, the *Landingpoint Finder* has been modified such that only 1 edge is pruned in every iteration. A graph of the data from the test can be seen in Figure 6.1, and the actual data can be found in Appendix F.

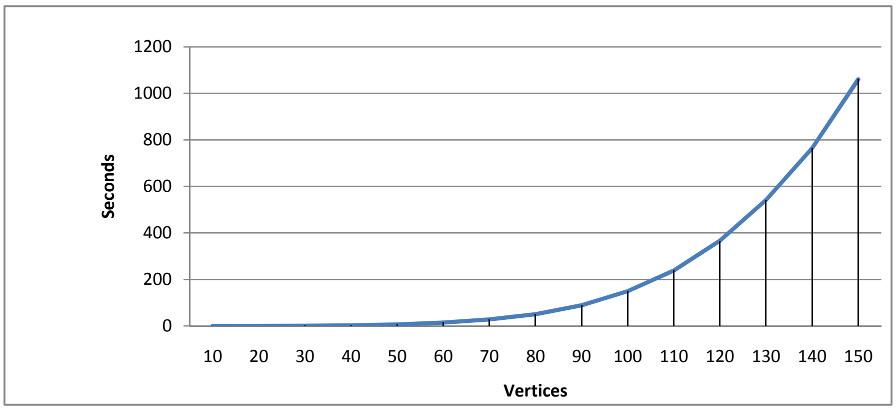

Figure 6.1: Performance of the Landingpoint Finder approach.

Looking at the results from the test, shown in Figure 6.1, the time it takes to find the landing point seems to roughly double for every 10 vertex. Initially, finding the traveling salesman path takes negligible time, but as the number of vertices increase, the NP-hardness of the problem starts showing. Therefore, as the number of vertices increase, the time to calculate the traveling salesman path starts being the main factor in finding the landing point. Because of that, it is important to ensure that as low a number of paths are calculated as possible. This is where our approach shows its strength. The data in Figure 6.2 shows the difference in the worst-case number of paths that have to be calculated.

Vertices:	10	20	30	40	50	60	70	80
LPFinder TSP Count:	90	380	870	1560	2450	3540	4830	6320
Exact TSP Count	1024	1048576	1,074E+09	1,1E+12	1,1259E+15	1,1529E+18	1,1806E+21	1,2089E+24

Vertices:	90	100	110	120	130	140	150
LPFinder TSP Count:	8010	9900	11990	14280	16770	19460	22350
Exact TSP Count	1E+27	1,27E+30	1,298E+33	1,329E+36	1,3611E+39	1,3938E+42	1,4272E+45

Figure 6.2: Comparison of worst-case number of TSP calculations.

By comparing the data in Figure 6.2, it is obvious that our approach requires significantly less traveling salesman calculations in order to find the landing point. Both approaches however, can be optimized further; the exact approach, as described in Section 4.2.3.1, and the prune approach by not calculating the traveling salesman path more than once per component. The incredibly large amount of subgraphs that has to be calculated using the exact approach has led us to not implement the exact approach. We have decided so, as we doubt that it would be possible to get any results within reasonable time with more than 20 vertices, and therefore the approach has no real use. Our decision is further supported by the fact that our approach requires more than 1000 seconds with 150 nodes, 22350 traveling salesman paths, whereas the exact would require more than 1 billion traveling salesman paths to be calculated at only 20 vertices.

This huge difference in number of traveling salesman paths to calculate, supports our belief in the pruning method as a viable approach to finding a good estimate to the optimal landing point in a short amount of time.

6.1.2 Score

To test how well the *Landingpoint Finder* performs regarding the score, it is necessary to have something to compare it to. To do this, the two methods *Barycentric* and *Gauss*, described in Section 4.1, have been implemented in *Jetpack Attack*, so the general functionality is the same, only the landingpoint calculations are different. Each of the three methods of finding a landing point is tested in three maps, with no opponents. The outcome of these tests will be three scores for each method, which can be compared to evaluate the *Landingpoint finder*. The maps are the official maps from Project Hoshimi, used in the final stages of the competition. Table 6.1 summarizes the tests, and the detailed results can be seen in Appendix E.

All	**Objectives Solved**	**Score**
Barycentric	18	14755
Gauss	16	17740
LPFinder	20	18570

Table 6.1: Cumulative Scores

As the table shows, the *Landingpoint Finder* scored a higher total than the other two methods, along with completing more objectives. If we look at the results in Appendix E, the *Landingpoint Finder* was beaten once on the score by a 100 points, but in general, the *Landingpoint Finder* always scores a high amount of points, compared to the others whose score fluctuates more. The reason for these fluctuations, is the inherent flaws in the methods, mentioned in Section 4.1. The results of the experiments performed on the *Landingpoint Finder*, show that our implementation fulfills its purpose. A landing point is found within a reasonable amount of time, and regarding the score, it also outperforms the other methods presented. This concludes the experiments on the *Landingpoint Finder*.

6.2 Mission Analyzer

This section describes the experiments performed on the mission analyzer, i.e. the Bayesian network described in Section 4.3. The basic idea behind the experiments, is to gather data from different missions, and then insert this as evidence into the network. This data taken from six official maps, used both in the later stages of the competition, and in a series of minor small competitions.

The data gathered from the different maps can be seen in Appendix G, and the results produced by inserting this evidence for the different maps, can be seen in Table 6.2.

Map	Factories	Hoshimi	Navigation	Unique Navigation
Round2-1	3.51	**39.96**	33.56	22.97
Round2-2	12.17	**64.16**	18.41	5.26
Round2-3	11.52	**60.76**	17.43	10.26
SC1	9.81	**51.71**	8.75	29.73
SC2	10.26	**54.07**	9.15	26.53*
SC3	21.12	**42.37**	12.16	24.36

Table 6.2: Bayesian Results

It should be noted that the maps *SC1*, *SC2* and *SC3* are older maps, which had a different version of the *Factory* objective, where all factories had to be destroyed, to receive points. The kill percentage value has therefore been set to *Medium* which was the value for the other three missions. The value marked with a '*', signifies that there was no *Unique Navigation* objectives in this map, so this should be disregarded. The chosen focus point for each map, has been marked with bold typeface. As mentioned in Section 4.3 and as the table shows, the *Hoshimi* focus point, is chosen for every map. The reason for this is, that in all maps, the number of hoshimi points is so high, that it is always possible to score a high number of points, just from placing needles at hoshimi points, and then filling them with OXY gas.

Because of these results, it was decided not to implement the Bayesian network in *Jetpack Attack*, because if there was not a radical change in the types of maps used, the outcome of the mission analysis would always be the same, i.e. that hoshimi points should be the main focus. Therefore, using the Bayesian network to analyze the mission, and assign priorities to different tasks, would just add an unnecessary computational overhead. In a game like Project Hoshimi where it is already necessary to use efficient techniques to accommodate the time-restricted, turn-based gameplay, it would be undesirable to introduce more overhead. Besides, focusing on hoshimi points also makes sense, since in every map seen so far, they have been the number one, undisputed point giver in the game. This concludes the experiments performed on the mission analyzer.

6.3 Competition

This section describes the results produced by *Jetpack Attack* in the official competition part of Project Hoshimi. The competition is structured as four rounds, where the teams are divided in to pools of four to six players, and the best in the pool goes through to the next round. The competitors in these four rounds, are the top three of each country, but since we were the only team from Denmark, we were automatically qualified for this stage.

The final results of the Project Hoshimi competition can be seen in [3], but it should be mentioned that *Jetpack Attack* finished in the top 20 of all competitors. If we compare it to the others which did not make it into top 10, based solely on points, *Jetpack Attack* finished 17th. Some of the reasons for not going further in the competition, became apparent after watching the replays of the matches we lost. First of all, two simple features were not implemented, which should have resulted in more points. The first is the fact that when *InjectorNeedle* placed on hoshimi points are destroyed, the hoshimi point is not added to the queue of hoshimi points, which do not have needles. Therefore, no new needle will be built on that point. The second problem occurs if the enemy has built his landing point close to, or on an OXY point. This point should be removed from the list of available OXY point, since the *Transporters* can not defend themselves, and if they keep trying to go to the OXY point in question, they will most likely be killed, and new ones will be built, thus preventing the *AI* bot from performing other tasks. These two errors in design could easily have been avoided, as they would most likely have been noticed, if we would have had more opponents to test our strategy on. Another thing which would have helped in scoring points, or at least prevented the opponent from scoring as many points, would have been to send out *Destroyers* or *Hunters* to hoshimi points, to guard them, and kill any enemy bots which might be there. This was one of the strategies of some of the opponents we encountered. Another thing keeping *Jetpack Attack* out of the top 10, was the quality of our opponents in the later rounds. Two of the teams we faced in the last pool, went in to the top six and thus won a spot in the final. These players have also participated in Project Hoshimi for several years, and as mentioned in Chapter 1, it was expected that these competitors would beat us.

This concludes the description of the experiments performed to evaluate the outcome of this project. The next chapter concludes upon the entire project as a whole.

Chapter 7

Conclusion

This chapter serves as an evaluation and conclusion upon this project, dealing with implementing a multi-agent AI, for Project Hoshimi. The important results will be presented and discussed, and the project will be concluded upon and evaluated as a whole. Finally, a discussion of what could be improved in *Jetpack Attack* is presented.

As described in the introduction to this report, this project is a general study in implementing a multi-agent AI system. Through both competing in the Imagine Cup, and also seeing some of the other competitors play, we have gained valuable insight in how multi-agent systems work, and what pitfalls and caveats this presents. One of the goals of this project was creating a competitive *AI* for the competition, and by finishing as one of the top 20 players, this goals has been fulfilled. It also further adds to the evaluation of the implementation, since one of the reasons for not advancing further in the competition, was a couple bugs, which could have been corrected, had we had more opponents to test the implementation on.

Another goal of this project was to design and implement solutions for the two focus points, i.e. finding a good landing point and being able to analyze any given mission, to derive a strategy for scoring the highest possible number of points. As for finding a good landing point, we have shown in Section 6.1, that our implementation of the *Landingpoint Finder* performs reasonably well both in terms of score and performance. In regards to the mission analysis, we have shown that our solution was not worthwhile, because of the lack of variation in the maps of Project Hoshimi. However, in a general game setting, similar to Project Hoshimi but with more variation in the missions, our method of analyzing the mission to find focus points, would be valid. The problem of gathering enough data for populating the CPTs, would still persist, but in most cases, games are not part of a competition, and it should therefore be easier to get players involved.

7.1 Future Work

Among the things which could be improved in *Jetpack Attack*, besides the obvious of fixing the bugs mentioned earlier, improving the current algorithm used for finding a landing point in one of them. Specifically, there are two main points, where it could be improved. One would be using a better algorithm for solving the TSP, and the other would be to improve the *Prune* algorithm, as it currently spends a lot time, searching every edge in the graph in each iteration. Instead, edges and their weights could be saved in a sorted list, to make selection of the highest weighted edge, faster. Another improvement could be taking airstreams into account, which would make the pathfinding more accurate and therefore also the landing point finder.

The optimal improvement would be developing an exact solution for finding the optimal subgraphs, which runs in reasonable time, compared to the brute-force approach. Whether a such method exists, is at the time of writing, unknown to the authors.

Chapter 8

Bibliography

[1] Margherita Barile. *Taxicab Metric*. http://mathworld.wolfram.com/TaxicabMetric.html. Online: 28/05-2008.

[2] Richard Clark. *Project Hoshimi*. http://www.project-hoshimi.com/. Online: 06/02-2008.

[3] Richard Clark. *Project Hoshimi 2008 Results*. http://project-hoshimi.com/2008/PH2008Results.aspx, 2008. Online: 31/05-2008.

[4] Microsoft Corporation. *Microsoft XNA Game Studio Contest*. http://dreambuildplay.com/main/default.aspx. Online: 30/05-2008.

[5] Olav Geil. *The Mathematical Foundation of A**. http://www.math.aau.dk/~olav/undervisning/dat06/astar.pdf, 2006. Online: 23/05-2007.

[6] Michael Hardy. *Gaussian Function*. http://en.wikipedia.org/wiki/Gaussian_function. Online: 31/05-2008.

[7] Canut Ki in club de go de Lyon. *Simulateur d'influence / Interactive go maps*. http://canut-ki-in.jeudego.org/simul_influence/, 2007. Online: 17/12-2007.

[8] Helge Langseth and Thomas D. Nielsen. *Classification using Hierarchical Naïve Bayes models*. http://www.cs.aau.dk/~tdn/papers/LangsethNielsen-HNB.pdf, 2006. Online: 27/05-2008.

[9] Microsoft. *Imagine Cup - Home*. `http://imaginecup.com/`, 2007. Online: 06/02-2008.

[10] et al Mike Dickheiser. *Game Programming Gems 6*. Charles River Media, 2006.

[11] RaptorXP. *Finding the good Injection Point (Part 1 & 2)*. `http://www.project-hoshimi.com/lstArticles.aspx?filter=dev`. Online: 31/05-2008.

[12] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[13] Eric W. Weisstein. *Gaussian Function*. `http://mathworld.wolfram.com/TaxicabMetric.html`. Online: 31/05-2008.

Appendix A

List of Acronyms

AI	Artificial Intelligence
CPT	Conditional Probability Table
FSM	Finite State Machine
SDK	Software Development Kit
TSP	Traveling Salesman Problem

Appendix B

Bot Characteristics

This appendix contains the template of each type of bot.

B.1 AI Bot

Characteristic	**Value**	**Actions**
ContainerCapacity	0	Self Destruct
CollectTransferSpeed	0	Move
Scan	5	Stop Moving
MaxDamage	0	Build Bot
DefenseDistance	0	
Constitution	50	
Shield	100	
Total	NA	

Table B.1: AI Template

B.2 Collector Bot

Characteristic	**Value**	**Actions**
ContainerCapacity	20	Self Destruct
CollectTransferSpeed	5	Move
Scan	5	Stop Moving
MaxDamage	5	Defend
DefenseDistance	12	Collect OXY
Constitution	50	Transfer OXY
Total	50	

Table B.2: Collector Template

B.3 Container Bot

Characteristic	**Value**	**Actions**
ContainerCapacity	60	Self Destruct
CollectTransferSpeed	5	Move
Scan	0	Stop Moving
MaxDamage	0	Collect OXY
DefenseDistance	0	Transfer OXY
Constitution	60	
Total	70	

Table B.3: Container Template

B.4 LPCreator Bot

Characteristic	**Value**	**Actions**
ContainerCapacity	0	Self Destruct
CollectTransferSpeed	0	Move
Scan	30	Stop Moving
MaxDamage	0	Open Landing Point
DefenseDistance	0	Close Landing Point
Constitution	20	
Total	40	

Table B.4: LPCreator Template

B.5 Explorer Bot

Characteristic	**Value**	**Actions**
ContainerCapacity	0	Self Destruct
CollectTransferSpeed	0	Move
Scan	30	Stop Moving
MaxDamage	0	
DefenseDistance	0	
Constitution	20	
Total	40	

Table B.5: Explorer Template

B.6 Needle Bot

Characteristic	**Value**	**Actions**
ContainerCapacity	100	Self Destruct
CollectTransferSpeed	0	Defend
Scan	10	
MaxDamage	5	
DefenseDistance	10	
Constitution	150	
Shield	100	
Total	150	

Table B.6: Needle Template

B.7 Blocker Bot

Characteristic	**Value**	**Actions**
ContainerCapacity	0	Self Destruct
CollectTransferSpeed	0	
Scan	10	
MaxDamage	0	
DefenseDistance	0	
Constitution	100	
Total	90	

Table B.7: Blocker Template

B.8 Wall Bot

Characteristic	**Value**	**Actions**
ContainerCapacity	0	Self Destruct
CollectTransferSpeed	0	
Scan	10	
MaxDamage	0	
DefenseDistance	0	
Constitution	100	
Total	90	

Table B.8: Wall Template

Appendix C

Conditional Probability Tables

C.1 Factories

Factories	Low	Medium	High
Low	0.05	0.1	0.7
Medium	0.25	0.8	0.25
High	0.7	0.1	0.05

Table C.1: FacCount CPT

Factories	Low	Medium	High
Low	0.7	0.1	0.05
Medium	0.25	0.8	0.25
High	0.05	0.1	0.7

Table C.2: FacScore CPT

Factories	Low	Medium	High
Low	0.05	0.1	0.7
Medium	0.25	0.8	0.25
High	0.7	0.1	0.05

Table C.3: FacNoOfBots CPT

Factories	Low	Medium	High
Low	0.7	0.1	0.05
Medium	0.25	0.8	0.25
High	0.05	0.1	0.7

Table C.4: FacKillPercent CPT

Strategy	Factories	Hoshimi	Navigation	Unique Navigation
Low	0.05	0.333333	0.333333	0.333333
Medium	0.15	0.333333	0.333333	0.333333
High	0.8	0.333333	0.333333	0.333333

Table C.5: Factories CPT

C.2 Hoshimi

Hoshimi	Low	Medium	High
Low	0.7	0.1	0.05
Medium	0.25	0.8	0.25
High	0.05	0.1	0.7

Table C.6: HoshimiScore CPT

Strategy	Factories	Hoshimi	Navigation	Unique Navigation
Low	0.333333	0.05	0.333333	0.333333
Medium	0.333333	0.15	0.333333	0.333333
High	0.333333	0.8	0.333333	0.333333

Table C.7: Hoshimi CPT

C.3 Navigation

Navigation	Low	Medium	High
Low	0.7	0.1	0.05
Medium	0.25	0.8	0.25
High	0.05	0.1	0.7

Table C.8: NavScore CPT

Navigation	Low	Medium	High
Low	0.05	0.1	0.7
Medium	0.25	0.8	0.25
High	0.7	0.1	0.05

Table C.9: NavNoOfBots CPT

Strategy	Factories	Hoshimi	Navigation	Unique Navigation
Low	0.333333	0.333333	0.05	0.333333
Medium	0.333333	0.333333	0.15	0.333333
High	0.333333	0.333333	0.8	0.333333

Table C.10: Navigation CPT

C.4 Unique Navigation

Unique Navigation	Low	Medium	High
Low	0.7	0.1	0.05
Medium	0.25	0.8	0.25
High	0.05	0.1	0.7

Table C.11: UNavDist CPT

Unique Navigation	Low	Medium	High
Low	0.7	0.1	0.05
Medium	0.25	0.8	0.25
High	0.05	0.1	0.7

Table C.12: UNavScore CPT

Strategy	Factories	Hoshimi	Navigation	Unique Navigation
Low	0.333333	0.333333	0.333333	0.05
Medium	0.333333	0.333333	0.333333	0.15
High	0.333333	0.333333	0.333333	0.8

Table C.13: UniqueNavigation CPT

Appendix D

Bayesian Network Values

D.1 Factories

Factories	Low	Medium	High
Score	Less than 1500	Between 1500 and 3000	Greater than 3000
No. of bots	Less than 7	Between 7 and 12	Above 12
Kill Percentage	50% - 55%	56%-60%	Above 60%

Table D.1: Factories Values

D.2 Hoshimi

Hoshimi	Low	Medium	High
Score	Less than 1500	Between 1500 and 3000	Greater than 3000

Table D.2: Hoshimi Values

D.3 Navigation

Navigation	Low	Medium	High
Score	Less than 1500	Between 1500 and 3000	Greater than 3000
No. of bots	Less than 5	Between 5 and 8	Greater than 8

Table D.3: Navigation Values

D.4 Unique Navigation

Unique Nav.	Low	Medium	High
Score	Less than 1500	Between 1500 and 3000	Greater than 3000
Distance	Less than 500	Between 500 and 1200	Greater than 1200

Table D.4: Unique Navigation Values

Appendix E

Score Experiment Results

The following shows the results of the experiments regarding the score of the *Landingpoint Finder*. The *Navigation*, *Factory* and *Other* columns, shows how many of the given type of objectives have been solved. *Navigation* covers both regular and unique navigation objectives. The top left corner of the tables, hold the names of the maps.

SC4	**Navigation**	**Factory**	**Other**	**Score**
Barycentric	4	1	1	5920
Gauss	1	1	1	2730
LPFinder	4	1	1	5820

Table E.1: SC4 Scores

Round2-2	**Navigation**	**Factory**	**Other**	**Score**
Barycentric	3	1	1	2300
Gauss	4	1	1	5360
LPFinder	5	1	1	5790

Table E.2: Round2-2 Scores

Round2-3	**Navigation**	**Factory**	**Other**	**Score**
Barycentric	5	1	1	6535
Gauss	5	1	1	6560
LPFinder	5	1	1	6960

Table E.3: Round2-3 Scores

Appendix F

LPFinder Performance Test

This example contains the data from the *Landingpoint Finder* performance test.

Vertices:	10	20	30	40	50	60	70	80
Seconds:	0,0174	0,1049	0,5965	2,2762	6,1705	14,0720	27,6506	50,1712

Vertices:	90	100	110	120	130	140	150
Seconds:	88,6033	149,3675	238,3566	366,2042	542,2150	764,7058	1060,2496

Figure F.1: Landingpoint Finder performance test data.

Appendix G

Bayesian Data

Round2-1	Value
FacScore	800
FacKillPercent	57%
FacNoOfBots	13
HoshimiScore	7920
NavScore	1600
NavNoOfBots	3
UNavDist	1270
UNavScore	500

Table G.1: Round2-1 Data

Round2-2	Value
FacScore	800
FacKillPercent	57%
FacNoOfBots	7
HoshimiScore	7920
NavScore	1600
NavNoOfBots	5
UNavDist	330
UNavScore	500

Table G.2: Round2-2 Data

Round2-3	Value
FacScore	800
FacKillPercent	57%
FacNoOfBots	7
HoshimiScore	9020
NavScore	1700
NavNoOfBots	4
UNavDist	699
UNavScore	500

Table G.3: Round2-3 Data

SC1	Value
FacScore	500
FacKillPercent	100%
FacNoOfBots	6
HoshimiScore	4400
NavScore	600
NavNoOfBots	6
UNavDist	2016
UNavScore	1000

Table G.4: SC1 Data

SC2	Value
FacScore	1000
FacKillPercent	100%
FacNoOfBots	7
HoshimiScore	6600
NavScore	1200
NavNoOfBots	5
UNavDist	0
UNavScore	0

Table G.5: SC2 Data

SC3	Value
FacScore	600
FacKillPercent	100%
FacNoOfBots	6
HoshimiScore	7260
NavScore	2000
NavNoOfBots	8
UNavDist	1722
UNavScore	600

Table G.6: SC3 Data