Multi-Agent Potential Field Based Architectures for Real-Time Strategy Game Bots

Johan Hagelbäck

© 2012-01-11 Johan Hagelbäck
School of Computing
Publisher: Blekinge Institute of Technology
Printed by Printfabriken, Karlskrona, Sweden 2012-01-11
ISBN 978-91-7295-223-2

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2012:02
ISSN 1653-2090
ISBN 978-91-7295-223-2

Multi-Agent Potential Field Based Architectures for Real-Time Strategy Game Bots

Johan Hagelbäck

School of Computing
Blekinge Institute of Technology
Sweden

Contact information:
Johan Hagelbäck
School of Computing
Blekinge Institute of Technology
371 79 Karlskrona
SWEDEN
email: `johan.hagelback@gmail.com`

“The weather outside is hostile with a slight chance of fog-of-war.”

– Medivacs in StarCraft 2

ABSTRACT

Real-Time Strategy (RTS) is a sub-genre of strategy games which is running in real-time, typically in a war setting. The player uses workers to gather resources, which in turn are used for creating new buildings, training combat units, build upgrades and do research. The game is won when all buildings of the opponent(s) have been destroyed. The numerous tasks that need to be handled in real-time can be very demanding for a player. Computer players (bots) for RTS games face the same challenges, and also have to navigate units in highly dynamic game worlds and deal with other low-level tasks such as attacking enemy units within fire range.

This thesis is a compilation grouped into three parts. The first part deals with navigation in dynamic game worlds which can be a complex and resource demanding task. Typically it is solved by using pathfinding algorithms. We investigate an alternative approach based on *Artificial Potential Fields* and show how an APF based navigation system can be used without any need of pathfinding algorithms.

In RTS games players usually have a limited visibility of the game world, known as *Fog of War*. Bots on the other hand often have complete visibility to aid the AI in making better decisions. We show that a Multi-Agent PF based bot with limited visibility can match and even surpass bots with complete visibility in some RTS scenarios. We also show how the bot can be extended and used in a full RTS scenario with base building and unit construction.

In the next section we propose a flexible and expandable RTS game architecture that can be modified at several levels of abstraction to test different techniques and ideas. The proposed architecture is implemented in the famous RTS game *StarCraft*, and we show how the high-level architecture goals of flexibility and expandability can be achieved.

In the last section we present two studies related to gameplay experience in RTS games. In games players usually have to select a static difficulty level when playing against computer opponents. In the first study we use a bot that during runtime can adapt the difficulty level depending on the skills of the opponent, and study how it affects the perceived enjoyment and variation in playing against the bot.

To create bots that are interesting and challenging for human players a goal is often to create bots that play more human-like. In the second study we asked participants to watch replays of recorded RTS games between bots and human players. The participants were asked to guess and motivate if a player was controlled by a human or a bot. This information was then used to identify human-like and bot-like characteristics for RTS game players.

ACKNOWLEDGMENTS

First, I would like to thank my main supervisor Dr. Stefan J. Johansson for invaluable support and guidance throughout the work. I would also like to thank Professor Paul Davidsson who was my secondary supervisor in my early work, and Professor Craig Lindley for taking over his role. In addition I would like to thank the community at AIGameDev.com and the users of BTHAI for comments and ideas on the project.

Last but not least this thesis wouldn't have been much without the support of my family, Maria and Mio, and our dog for forcing me to take refreshing walks every day.

Karlskrona, December 2011
Johan Hagelbäck

PREFACE

This thesis is a compilation of nine papers. The papers are listed below and will be referenced to in the text by the associated Roman numerals. The previously published papers have been reformatted to suit the thesis template.

I. J. Hagelbäck and S. J. Johansson (2008). Using Multi-agent Potential Fields in Real-time Strategy Games. In L. Padgham and D. Parkes editors, Proceedings of the Seventh International Conference on Autonomous Agents and Multi-agent Systems (AAMAS).

II. J. Hagelbäck and S. J. Johansson (2008). Demonstration of Multi-agent Potential Fields in Real-time Strategy Games. Demo Paper on the Seventh International Conference on Autonomous Agents and Multi-agent Systems (AAMAS).

III. J. Hagelbäck and S. J. Johansson (2008). The Rise of Potential Fields in Real Time Strategy Bots. In Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE).

IV. J. Hagelbäck and S. J. Johansson (2008). Dealing with Fog of War in a Real Time Strategy Game Environment. In Proceedings of 2008 IEEE Symposium on Computational Intelligence and Games (CIG).

V. J. Hagelbäck and S. J. Johansson. A Multi-agent Potential Field based bot for a Full RTS Game Scenario. In Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2009.

VI. J. Hagelbäck and S. J. Johansson. A Multiagent Potential Field-Based Bot for Real-Time Strategy Games. International Journal of Computer Games Technology, vol. 2009, Article ID 910819, 10 pages. doi:10.1155/2009/910819

VII. J. Hagelbäck. An expandable multi-agent based architecture for StarCraft bots. Submitted for publication.

VIII. Johan Hagelbäck and Stefan J. Johansson. Measuring player experience on runtime dynamic difficulty scaling in an RTS game. In Proceedings of 2009 IEEE Symposium on Computational Intelligence and Games (CIG), 2009.

IX. Johan Hagelbäck and Stefan J. Johansson. A Study on Human like Characteristics in Real Time Strategy Games. In Proceedings of 2010 IEEE Conference on Computational Intelligence and Games (CIG), 2010.

The author of the thesis is the main contributor to all of these papers.

CONTENTS

CHAPTER
ONE

INTRODUCTION

Real-Time Strategy (RTS) games is a sub-genre of strategy games which is runnig in real-time, typically in a war setting. The player controls a base with defensive structures that protect the base, and factories that produce mobile units to form an army. The army is used to destroy the opponent(s) units and bases. The genre became popular with the release of *Dune II* in 1992 (from Westwood Studios) and *Command & Conquer* 1995 (also from Westwood Studios). The game environment can range from medieval (*Age of Empires 2*), fantasy (*Warcraft II* and *III*), World War II (*Commandos I* and *II*), modern (*Command & Conquer Generals*) to science fiction (*StarCraft*, *StarCraft 2* and *Dawn of War*).

RTS games usually use the control system introduced in *Dune II*. The player select a unit by left-clicking with the mouse, or click and drag to select multiple units. Actions and orders are issued by right-clicking with the mouse. This control system is suitable for mouse and keyboard, and the genre has had little success at consoles even though several games in the *Command & Conquer Generals* series have been released for both PC and consoles. Since the release of Playstation 2 and Xbox in early 2000 console games have largely outnumbered the PC game sales. In 2005 the total PC game sales in US was $1.4 billion and games for consoles and handheld devices sold for $6.1 billion (Game Sales Charts, 2011). With the reduced interest in PC games and RTS already being a small genre the interest in such games has been low. The game genre received a huge popularity boost in 2010 with the awaited release of *StarCraft 2*, which was reported to have sold over 1.5 million copies the first 48 hours after release (StarCraft 2 Sales, 2010). RTS games are very suitable for multi-player and their complex nature have made them very popular in competitions, where *Warcraft III*, *StarCraft* and *StarCraft 2* are big titles at the E-sport scene (Electronic Sports, 2011).

From a general perspective, the gameplay can be divided into a number of sub-tasks (Buro & Furtak, 2004):

- *Resource gathering*. Resources of one or more types must be gathered to pay for buildings, upgrades and units. This usually means that a number of worker units have to move from the command center to a resource spot, gather resources, then return to the command center to drop them off. The command center is the main structure of a base and is usually used to create new workers and act as a drop-off point for resources. Games usually have a limitation of how many workers that can gather from a single resource patch at a time, which gives an upper bound of the income rate from each resource area. By expanding to build bases in new areas the player gain control of more resources and increase the total income, but makes it more difficult to defend the own bases. When expanding to a new area the player must initially spend an often significant amount of resources to build a new command center and possibly defensive structures, making the decision of when and where to expand somewhat difficult.

- *Constructing buildings*. Creating new buildings takes time and cost resources. There are four major types of buildings: command centers (drop-off points for resources), unit production buildings (creates new units), tech buildings (contains upgrades for units and can unlock new powerful units or buildings) and defensive structures (missile towers, bunkers etc.). Most games have a limit of the number of units that can be created. In for example *StarCraft* this is handled by a fifth type, supply buildings. Each supply building can support a number of units, and the amount of supply buildings sets an upper bound of how many mobile units that can be created. New supply buildings have to be constructed to increase this limit.

- *Base planning*. When the player has paid for a new building, he/she must also decide where to place it. Buildings should not be spread out too much since it will make the base hard to defend. They should not be placed too close either since mobile units must be able to move through the base. Expensive but important buildings should be placed where they have some protection, for example near map edges or at the center of a base. Defensive and cheap buildings should be placed in the outskirts. The player have a limited area to build his base on, and that area is usually connected to the rest of the map with a small number of chokepoints from where the enemy can launch an attack. The player must also be prepared for aerial attacks from any direction except map edges.

- *Constructing units*. Units cost time and resources to build and there are often a limit of how many units each player can have. This, combined with the fact that there are often lots of different types of units available, makes it rather difficult to decide which unit to construct and when. An important aspect of RTS games is balance. Balance does not mean that all the races in the game must have the same type of units, but rather that all races have a way of defeating the other races by the clever use of different types of units. Each unit has different strengths and weaknesses, and they often counter other units in a rock-paper-scissor like fashion. There are several good examples of this in *StarCraft*. Terran Siege Tanks are very powerful units that can attack at long range but they have a number of weaknesses; 1) they must be in siege mode to maximize damage and they cannot move when sieged, 2) they cannot attack air units, and 3) they cannot attack at close range and are therefore very vulnerable to close-range units like Protoss Zealots. Siege Tanks alone can quite easily be countered, but when combined with other units they are extremely deadly. In practice it is almost impossible to create a perfectly balanced game, and all RTS games are more or less unbalanced. *StarCraft* with the expansion *Broodwar* is generally considered to be extremely well balanced (Game Balance, 2011).

- *Tech-tree*. The player can invest in tech buildings. Once constructed, one or more upgrades are available at the building. For a quite significant amount of resources and time the player can research an upgrade that will affect all units of a specific type. Upgrades can make a unit type do more damage or give it more armor, or it can give a completely new ability to the unit type. Tech buildings and upgrades can also unlock new production buildings to give access to more powerful units. In the *StarCraft* example the player can research the Vehicle Weapons upgrade to increase the damage of all vehicle units. He/she can also research the Siege Mode upgrade to unlock Siege Mode ability for Terran Siege Tanks, and Protoss players must build a Templar Archives to unlock Templar units at the Gateway production building. Since technology cost resources and time, the player must carefully consider which upgrades and techs to research. If for example a Terran player rarely or never use Vultures in his playstyle, it is no use to research the technology Spider Mines since it only affects Vultures.

The endless possibilities, the richness and complexity of the game world make RTS games very challenging for human players. It also makes it very difficult and time consuming to design and develop computer players (usually referred to as bots) for such games. The bots have to be able to navigate an often large amount of units in a highly dynamic, large game world. Units must be able to find the way to their destination, and move there without colliding with terrain or other units or buildings. The bot also have to be able to construct and plan base(s), decide which units to use to counter the units used by the enemy, research suitable upgrades, and make complex tactical decisions such as how to defend own base(s) or from where to best launch an attack at enemy bases. The real-time aspect of the games also makes performance in terms of quick decision making a very important aspect for bots. In this thesis we will investigate several problems when designing and implementing bots for RTS games. These are:

- *Navigation*. The navigation of units in strategy games are usually handled with path planning techniques such as A*. We will investigate an alternative approach for navigation based on Artificial Potential Fields.

- *Architectures*. We propose a general Multi-Agent Artificial Potential Field based architecture that is implemented and tested in several games and scenarios.
- *Gameplay experience*. A goal for an RTS game bot is often to try to play as human-like as possible. We will investigate what human-like features mean in RTS games, and if pursuing these features enhance the gameplay experience.

1.1 Background and Related Work

Background and Related Work is divided into three parts. The first part focuses on design and architectures for RTS game bots. The second part is about navigation in virtual worlds with pathfinding algorithms and *Artificial Potential Fields*, followed by a discussion of complete versus limited visibility of the game world. The third part discusses gameplay experience and human like behavior in games. These parts cover the research questions described in chapter 1.2.

1.1.1 Design and Architectures

Playing an RTS game is a complex task, and to handle it a bot is usually constructed with several modules that each handles a specific sub problem. A common architecture is the command hierarchy, described by for example Reynolds (2002). It is a layered architecture with four commanders; *Soldier*, *Sergeant*, *Captain* and *Commander*. Each layer has different responsibilities and available actions. The Commander layer takes strategic decisions on a very high level, for example when and from where to launch an attack at enemy base(s). To execute actions the Commander gives orders to the Captains (the architecture can have any number of lower level commanders in a tree-like structure, with the Commander as the root node). The Captains re-formulate the orders from the Commander and gives in turn orders to their Sergeants (which usually is a leader of a squad of units). The Sergeants control their Soldiers, which is usually a single unit in the game, to complete the orders from the Captain. The lower the level, the more detailed actions are issued. The communication between the layers is bi-directional; the higher levels issue orders to lower levels and the lower levels report back status information and other important things such as newly discovered enemy threats. Figure 1.1 shows an overview of a general command hierarchy architecture (Reynolds, 2002).

1.1.2 Navigation

Navigation of units in RTS games is a complex task. The game worlds are usually large, with complex terrain and regions that can only be accessible through narrow paths.

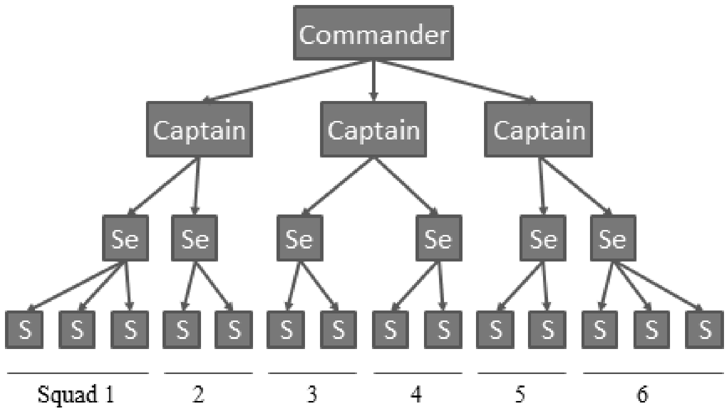


Figure 1.1: *The Command Hierarchy architecture.*

Some maps can also have islands which can only be reached by aerial units. An often very large amount of units must be able to find paths to their destinations in the static terrain. This is typically solved using a pathfinding algorithm, of which A* is the most common. A*, first described by Hart et.al. in 1968, has been proven to expand equal or less number of nodes when searching for a path than any other algorithm (Hart, Nilsson, & Raphael, 1972). Although A* solves the problem of finding the shortest path between two locations in a game world, we still have to deal with the highly dynamic properties of an RTS game. It takes some time to calculate a path with A*, and even longer time for a unit to travel along the path. During this time many things can happen in a dynamic world that makes the path obsolete. Re-planning the full or part of a path when a collision occurs is an option, but if several units re-plan at the same time this can cause deadlocks. Think about when you go straight towards someone on the sidewalk. You take a step to the side to avoid bumping into them, they take a step to the same side, you smile a bit, both steps back and this continues until you or the other person waits.

Extensive work has been made in optimizing A* to deal with these issues and improve performance of pathfinding in games. Higgins (2002) describes some tricks that were used to optimize the pathfinding engine in the RTS game *Empire Earth*. Demyen and Buro (2008) address the problem of abstracting only the information from the game world that is useful for the pathfinder engine by using triangulation techniques. In Koenig and Likhachev (2006) an approach for improving the performance of A* in adaptive game worlds by updating heuristics in nodes based on previous searches is described. Additional work on adaptive A* can be found in Sun, Koenig, and Yeoh (2008) where the authors propose a Generalized Adaptive A* method that improve performance in game worlds where the action cost for moving from one node to another can increase or decrease over time.

When using pathfinding algorithms in dynamic worlds it is quite common to use local obstacle avoidance, both to solve and detect collisions. *Artificial Potential Fields* is one technique that successfully has been used for obstacle avoidance in virtual worlds. It was first introduced by Khatib (1986) for real-time obstacle avoidance for mobile robots. It works by placing attracting or repelling charges at important locations in the virtual world. An attracting charge is placed at the position to be reached, and repelling charges are placed at the positions of obstacles. Each charge generates a field of a specific size. A repelling field around obstacles are typically small while the attracting field of positions to be reached has to cover most of the virtual world. The different fields are weighted and summed together to form a total field. The total field can be used for navigation by letting the robot move to the most attracting position in its near surroundings. Many studies concerning potential fields are related to spatial navigation and obstacle avoidance, for example the work by Borenstein and Koren (1991) and Massari, Giardini, and Bernelli-Zazzera (2004). Alexander (2006) describes the use of fields for obstacle avoidance in the games *Blood Wake* and *NHL Rivals*. Johnson (2006) describes obstacle avoidance using fields in the game *The Thing*.

Besides obstacle avoidance combined with pathfinding, there have been few attempts to use potential fields in games. Thurau, Bauckhage, and Sagerer (2004b) have developed a bot for the first-person shooter game *Quake II* that learns reactive behaviors from observing human players by modifying weights of fields. Wirth and Gallagher (2008) used a technique similar to potential fields in the game *Ms.Pacman*. Potential fields has also been used in robot soccer (Johansson & Saffiotti, 2002; Röfer et al., 2004).

Figure 1.2 shows an example of how Potential Fields (PFs) can be used for navigation in a game world. A unit in the lower left corner moves to its destination at E. The destination has an attractive charge (light areas) that gradually fades to zero (dark areas). Mountains (black) and two obstacles (white circles) generate small repelling fields (darker areas) for obstacle avoidance.

A unit navigating using PFs only looks one step ahead, instead of planning a full path like pathfinding algorithms usually do. This makes PFs naturally very good at handling dynamic game worlds since the fields are updated every time a unit moves or a new obstacle is discovered. There are however a number of difficulties that have to be addressed when using PFs:

- Units navigating using PFs may get stuck in local optima. This can happen when a unit moves into a dead end (e.g. inside a U-shaped obstacle). Although there exist methods to solve this, many of them are based on a detection-recovery mechanism which takes more time than finding the shortest path directly with a pathfinding algorithm (Borenstein & Koren, 1989).

- Performance issues. To calculate how multiple fields affect all positions in a game world requires either lots of CPU time (if the fields are generated at run-time) or

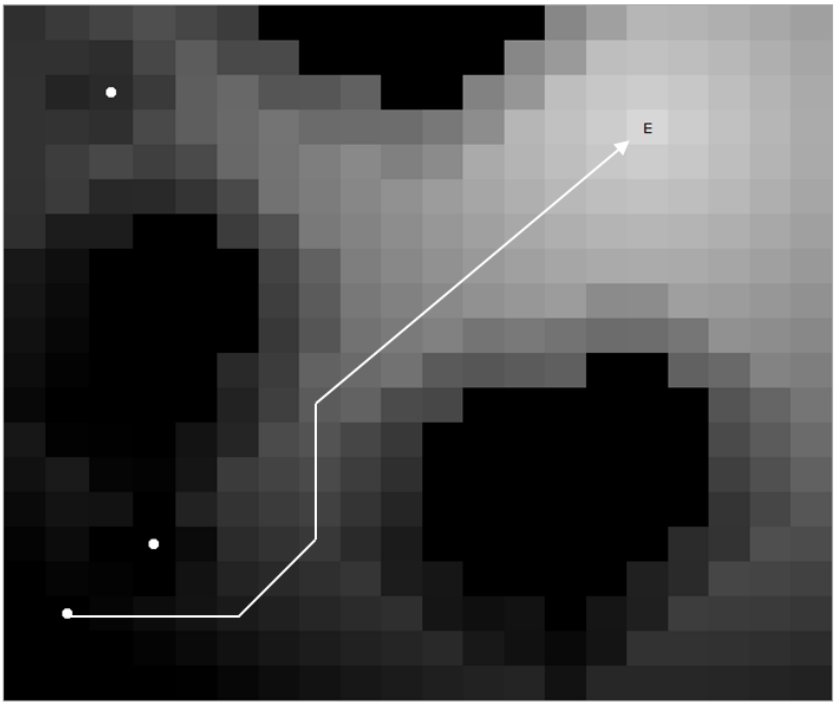


Figure 1.2: *Example of PFs in a virtual game world.*

lots of memory (if the fields are pre-calculated and stored in grids). Developers must be careful when implementing PF based solutions to not use up too much resources.

- PF based solutions can be difficult to tune and debug. When using a pathfinding algorithm you will always find the shortest or near shortest (some algorithms cannot guarantee optimality) path between two positions, assuming your pathfinding algorithm is correctly implemented. If the path looks weird, you start debugging the pathfinder. A unit navigating using potential fields is affected by multiple fields generated by many objects in the game world, and it can require lots of tuning to find the correct balance between the different fields. It can also be difficult to find the reason behind strange paths. Is it due to a problem with a single field or due to a combination of several fields that causes the error? A visual field debugging system can however help solving most of these issues.

- PF based solutions can be less controllable than traditional solutions. Units navigating using pathfinding algorithms always follow their paths. If something goes wrong, it is either a problem with the pathfinder or the navigation mesh (the internal representation of the game world used by the pathfinder). As in the previous paragraph, units navigating using potential fields are affected by multiple fields and it can be difficult to predict how the design of a field or the collaboration between multiple fields will affect units in all situations. On the other hand, micro-managing several units in narrow passages is probably more controlled with a potential field based solution.

In RTS games the player usually only has a limited view of the game world. Unexplored areas are black and the player does not know anything about them. Areas where the player has units or buildings are completely visible and the player sees everything that happens there. Areas that previously have been explored but are currently not within visibility range of a unit or building are shaded and only show buildings that were present last time the area was visited. The player knows what the static terrain looks like, but cannot see if any enemy unit is in that area. This limited visibility is usually referred to as *Fog of War* or FoW. Figure 1.3 shows a screenshot from *StarCraft* displaying Fog of War.

It is quite common for RTS game bots to have complete visibility of the game world in contrast to the limited view a human player has. The purpose is that the more information the bot has, the better it can reason about the enemy and make intelligent decisions. Some people think this is fine. If it makes a bot more interesting and challenging to play against for a human player, why not give it access to more information than the player has? Some people think the bot is "cheating". The player cannot surprise the enemy by doing something completely unexpected since the bot sees everything the human player does. According to Nareyek (2004), cheating is *"very annoying for the player if discovered"* and he predicts the game AIs to get a larger share of the processing power in the future which in turn may open up for the possibility to use more sophisticated game AIs.

1.1.3 Gameplay experience

The goal of playing a game is to have fun. An exception is serious games but they will not be dealt with in this thesis. If a bot needs to "cheat" to be interesting and challenging enough for even expert players to enjoy the game, why not let it do that? The problem is; what makes a game fun, and more importantly how can we measure it? There are several different models of player enjoyment in computer games. Some well-known examples are the work of Malone in the early 80's on intrinsic qualitative factors for engaging game play (Malone, 1981a, 1981b), and the work of e.g. Sweetster and Wyeth on the Gameflow model (Sweetster & Wyeth, 2005).

A common way is to make bots or computer controlled characters (NPCs) more interesting by trying to make them behave more humanlike. Soni and Hingston (2008) let human players train bots for the first-person shooter game *Unreal Tournament* by

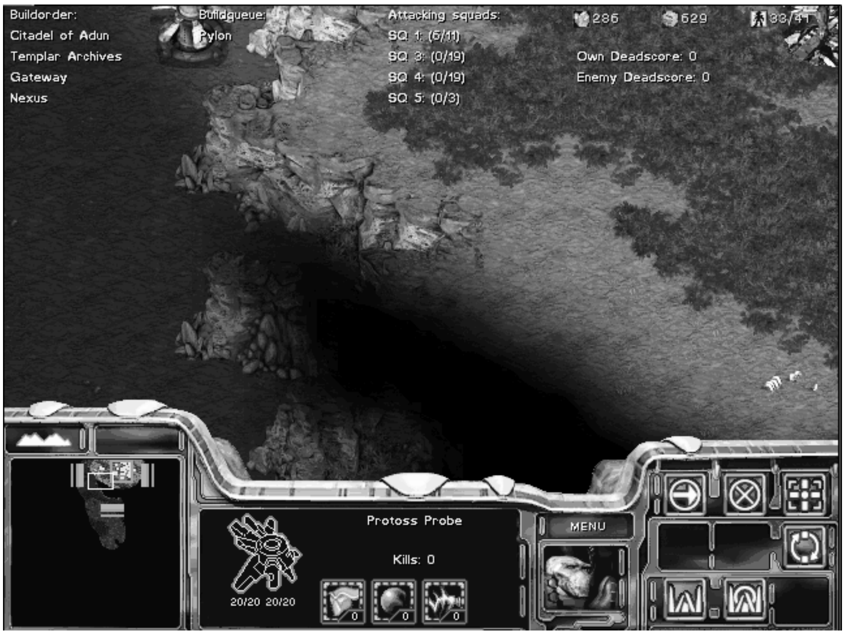


Figure 1.3: *A screenshot from StarCraft displaying Fog of War. The square in the lower left corner shows a map of the whole game world. Black areas are unexplored. Shaded areas have previously been explored but currently are not in visibility range of any unit or building.*

using neural networks. They conclude that the trained bots are more humanlike and were clearly perceived as more fun than coded rule-based bots. Yannakakis and Hallam (2007) mentions however that humanlike computer opponents does not always have to be more fun. Freed et al. (2007) have made a survey to identify the differences between human players and bots in *StarCraft*. They conclude that more humanlike bots can be valuable in training new players in a game or provide a testing ground for experienced players testing new strategic ideas.

Characters in games does not have to be intelligent. Developers and AI designers have to focus on what Scott (2002) defines as "The Illusion of Intelligence". Bots or computer controlled characters only have to do reasonably intelligent actions under most circumstances in a specific game to appear intelligent.

1.2 Research Questions

The main purpose of this thesis is to evaluate if potential fields is a viable option for navigating units in RTS games, and to design and evaluate multi-agent potential field (MAPF) based architectures for RTS games. We will investigate how well a potential field based navigation system is able to handle different RTS game scenarios, both in terms of performance in winning games against other bots but also performance in terms of computational resources used. We will also develop some guidelines for designing MAPF based architectures and evaluate them in different scenarios. At last we will perform some studies regarding player experience and human-like behavior for RTS game bots.

The following research questions are addressed:

RQ1. How does a MAPF based bot perform compared to traditional solutions?

This question is answered by studying performance of MAPF based bots in different RTS games and scenarios. It involves both performance in terms of playing the game well and defeat the opponents, and performance in terms of computational resources used.

RQ2. To what degree is a MAPF based bot able to handle incomplete information of a game world?

RTS game bots often "cheat" in the sense that they have complete visibility of the game world in contrast to the limited view a player has. The reason is often to give the bot more information than a player to be able to take better decisions and make the bot more interesting and challenging to play against. We will investigate how a MAPF based bot is able to handle a limited view of the game world, i.e. Fog of War, in an RTS scenario compared to bots with complete visibility.

RQ3. How can a MAPF based RTS bot architecture be designed to support flexibility and expandability?

This question is answered by designing and implementing a bot architecture that fulfills a number of flexibility and expandability requirements. Example of requirements are ability to play several races/factions available in a game, ability to modify or exhange logic at different levels of abstraction, and ability to play on different maps.

RQ4. What effects does runtime difficulty scaling have on player experience in RTS games?

When playing 1v1 games against a bot in most RTS games, the player have to manually select a difficulty level based on his/her experience and knowledge of the game. We will investigate the effects runtime difficulty scaling has on player experience factors such as challenge, entertainment and difficulty. Runtime difficulty scaling means that we adapt the performance in terms of playing the game well at runtime based on an estimation of how good the human player is.

RQ5. What are important characteristics for human-like gameplay in RTS games?

A common goal to improve gameplay experience and the fun factor of a game is to make bots and computer controlled characters behave more human-like. To do this the game designers and programmers must know what defines a human player in a specific game genre. We will perform a study to find human-like characteristics of players in RTS games.

1.3 Research Methods

RQ1 and RQ2 have been answered using a quantitative approach. We have designed and implemented a MAPF based bot for two scenarios in the open-source RTS game engine ORTS. The first scenario is what we refer to as Tankbattle. In the scenario each player has a fixed number of units (tanks) and buildings, and no production or research can be made. The winner is the first to destroy all buildings of the opponent. The second scenario is referred to as Full RTS. In this scenario each player starts with a number of workers and a command center, and has to construct buildings and units to be able to defeat the opponent. The bot has been tested in a yearly ORTS competition organized by the University of Alberta. We believe tournaments are a good test bed for a number of reasons; i). They are competitions and opponent bots will do their best to defeat us. ii). It is a standardized way of benchmarking performance of different solutions. iii). Tournaments are run by a third party which assures the credibility. In addition to the annual tournaments we have used bots from earlier tournaments as opponents in experiments to test new ideas and changes.

Although ORTS has several similarities with commercial RTS games, it is very much simplified. There are only a limited number of units (tanks and marines), few buildings (command center, barracks to produce marines and factories to produce tanks), and no real tech tree (only restriction is that the player must have a barrack to build a factory).

With the release of the *BWAPI* project in November 2009 it became possible to develop bots for the very well-known commercial game *StarCraft* and the *Broodwar* expansion (BWAPI, 2009). This is very interesting from a research perspective since *StarCraft* is the most famous RTS game ever released, it is known to be extremely well balanced, it has all the elements of a modern RTS game, and it is widely used in E-sport tournaments. The MAPF based bot was adapted to and implemented in *StarCraft*, and the resulting bot has been released as open-source under the project name BTHAI at Google Code[1]. The annual ORTS tournament has now been replaced by a *StarCraft* tournament, in which BTHAI has participated in three times.

The main goal of the *StarCraft* bot was however not to win as many games as possible in bot tournaments, but rather to show how a MAPF based bot architecture can support flexibility and expandability. This is addressed in RQ3, which has been answered

[1]http://code.google.com/p/bthai

with a proof of concept approach. We have designed and implemented the bot, and have shown that it supports a number of requirements related to flexibility and expandability.

RQ4 and RQ5 have been answered using an empirical approach. People participanting in the experiments have been asked to fill in questionnaires, and we have collected and grouped the data to form conclusions.

1.4 Contributions

In this section we address each research question and, in the process, summarize the included papers.

1.4.1 RQ1: How does a MAPF based bot perform compared to traditional solutions?

RQ1 is addressed in Papers I, II, III and VI. In Paper I we present a six-step methodology for designing a PF based navigation system for a simple RTS game. The methodology was evaluated by developing a bot for the Tankbattle scenario in the open-source game engine ORTS. Tankbattle is a two-player game where each player has 50 tanks and 5 command centers. No production can be done, so the number of tanks and buildings are fixed. To win the game a player has to destroy all command centers of the opponent. The static terrain and location of command centers are generated randomly at the start of each game. Ten tanks are positioned around each command center building. The bot participated in 2007 years' ORTS tournament where bots compete against bots in different scenarios. In Paper III several weaknesses and bugs in the bot were identified, and a new version was created.

Paper II is a demo paper describing a demonstration of the improved bot described in Paper III.

Paper VI is mostly a summary of Papers I, II and III. The contribution in this paper is how the MAPF based navigation systems can handle a resource gathering scenario. In this scenario each player has a command center and 20 workers. The workers shall move to resource patches, gather as much resources they can carry, and return to the command center to drop them off. In for example *StarCraft* workers are transparent when gathering resources and no collision detection needs to be handled. In this scenario all workers must avoid colliding with other own workers. The bot participated in the Collaborative Resource Gathering scenario in 2008 years' ORTS tournament. The winner is the bot which has gathered the most resources during a fixed game length.

Our conclusion is that MAPF based bots is a viable approach in some RTS scenarios being able to match and surpass the performance of more traditional solutions.

1.4.2 RQ2: To what degree is a MAPF based bot able to handle incomplete information of a game world?

RQ2 is addressed in Paper IV where we show how a MAPF based bot can be modified to handle incomplete information of the game world, i.e. Fog of War (FoW). We conclude that a bot without complete information can, in some scenarios, perform equally well or even surpass a bot with complete information without using more computational resources. Even if this suprisingly high performance was true in the game and scenario used in the experiments it is probably not valid for other games and scenarios. Still a potential field based bot is able to handle Fog of War well.

1.4.3 RQ3: How can a MAPF based RTS bot architecture be designed to support flexibility and expandability?

RQ3 is addressed in Paper VII where we show how a Multi-Agent Potential Field based bot for the commercial RTS game *StarCraft* can be designed to support high level architectural goals such as flexibility and expandability. In order to evaluate this these abstract goals were broken down to a number of requirements:

- The bot shall be able to play on a majority of *StarCraft* maps. Completely island-based maps without ground paths between starting locations are currently not supported.
- The bot shall be able to play all three races (Terran, Protoss and Zerg).
- High-level and low-level tactics shall be separated.
- Basic functions like move/attack/train unit shall work for all units without adding unit specific code.
- It shall be possible to micro-manage units by adding specific code for that unit type.
- Units shall be grouped in squads to separate squad behavior from single unit behavior.
- The bot shall be able to handle different tactics for different player/opponent combinations.

In the paper we used a modular multi-agent architecture with agents at different levels of abstraction. Agents at the lowest level were controlling single in-game units and buildings, while agents at the highest level handled tasks such as build planning, economy and commanding groups of combat units.

The requirements were evaluated in a proof-of-concept manner. We showed in different use cases and game play scenarios that all of the above mentioned requirements were met, therefore we conclude that the higher level goals of flexibility and expandability are met.

1.4.4 RQ4: What effects does runtime difficulty scaling have on player experience in RTS games?

RQ4 is addressed in Paper VIII. In the paper we have performed a study where human players were to play against one of five different bots in the ORTS game. The different bots originated from one bot and their difficulties were scaled down. The difficulty setting could be either static (same difficulty level in the whole game) or dynamic (difficulty changed depending on how well the human player performs). The bot versions used are:

- Static with medium difficulty.
- Static with low difficulty.
- Adaptive with medium difficulty. Difficulty rating changes slowly.
- Same as previous version, but drops difficulty to very low in the end of a game to let the human player win.
- Adaptive with medium difficulty. Quick changes in difficulty rating.

Each human player played against one random bot version and was asked to fill in a questionnaire after the game. The goal of the questionnaire was to find differences in enjoyment of playing against the bot, difficulty of winning against the bot and variation in the bots' gameplay.

1.4.5 RQ5: What are important characteristics for human-like gameplay in RTS games?

RQ4 is addressed in Paper IX. In this paper we performed a study aiming to give an idea of human-like characteristics of RTS game players. In the study humans were asked to watch replays of *Spring* games and decide and motivate if the players were human or bots.

To generate replays two different bots for the *Spring* game were developed. One bot uses an early tank rush tactic, while the other builds a large and strong base before attacking the enemy. Each bot comes in three versions where the pace of how often actions are performed is fast, slow or medium. In some replays bot played against bot, and in some replays humans against bot. In total 14 replays were generated and each participant in the study were asked to watch a randomly chosen game and fill in a questionnaire. In total 56 persons participated in the study.

1.5 Discussion and Conclusions

The discussion is divided into the three parts *Potential Fields*, *RTS game architectures* and *Gameplay experience in RTS games*.

1.5.1 Potential Fields

First we will discuss the previosly defined difficulties that have to be addressed when using potential fields in games.

Units navigating using PFs may get stuck in local optima

In Paper II we described how pheromone trails can be used to solve many local optima issues. Still a navigation system only based on potential fields can still have difficulties in more complex maps with many chokepoints, islands, and narrow paths. In Paper VII we use a navigation system that combines potential fields with pathfinding algorithms. It uses pathfinding when moving over long distances, and potential fields when getting close to enemy units or buildings. This solves almost all local optima problems in games. One of the major advantages of a potential field based system is that if the fields are modeled with the most attracting position at the maximum shooting distance of a unit, own units surround the enemy and weak units with strong firepower such as artillery tanks are kept in the back. This works well even in the combined approach.

Performance issues

In Papers I and III we show that a potential field based bot for the ORTS game can be implemented and run on the same hardware as other bots based on more traditional approaches without having computational issues. To investigate this in more detail we have implemented a tool that measures the time spent on updating potential fields in an RTS like scenario.

The tool uses a map of 64x64 terrain tiles where each tile is 16x16 positions, in total 1024x1024 positions. The map has some impassable terrain. The own player has 50 units to control and calculate potential fields for, the opponent has 50 units (that each generate a field) and there are also 50 neutral moving objects (which own units should avoid colliding with by using small repelling fields). The tool is running on a laptop with an Intel Core 2 Duo T7400 2.16 GHz CPU, 2 GB of RAM and Windows XP Pro 32 bit Servicepack 3.

We only calculate the potentials for positions that can be reached by one or more units. If each own unit can reach 32 different positions each frame we can greatly reduce the number of calls to the potential fields function. If we assume units always move at max speed we can further reduce the action space to 10 (9 directions in full speed + idle). This took 133ms per frame to complete.

So far we have calculated the potential field values generated by units and terrain each frame. If the terrain is static we can pre-calculate the terrain fields and store them in a grid (2-dimensional array). By doing this we sacrifice some memory resources to speed up computation. The tests using the tool showed an average frame time of 14ms.

Calculating the Euclidean distance between two objects in the game world takes some time since a square root operation is needed. Another improvement is to estimate the distance between an own unit and all other objects in the game world using the faster Manhattan distance calculation[2]. If the estimated distance is more than the maximum size of the field generated by an object times 1.42 (in worst case MH distance overestimates the Euclidean distance with $\sqrt{2}$) the object is too far away and will not affect the total field around the current unit. Therefore no Euclidean distance and potential field value calculation is needed. This reduces the average frame time to 12.8ms.

It is also possible to spread out the computation over several frames. We might not need to calculate new actions for each unit every frame. If we for example can reach the same performance in terms of how well the bot plays the game by choosing actions every 5:th frame instead of every frame, the total time spent on updating the fields would be 3-4ms per frame.

Paper VII also shows that a potential field based navigation system, although combined with pathfinding, works well even in more complex games like *StarCraft*.

PF based solutions can be difficult to tune and debug

Potential field based navigations system can be implemented using simple architectures and algorithms. Tuning can however be difficult and time consuming. The shape and weights of the fields surrounding each type of game object often have to be designed manually, although many object types share the same shape with the most attractive position at the maximum shooting distance from an enemy unit.

The value of weights, i.e. how attractive a field generated by an object is, can often be determined by the relative importance between different objects. For example a Protoss High Templar is a weak unit with very powerful offensive spells. It should be targeted before most other Protoss units such as Dragoons, and should therefore have a more attractive field than Dragoons.

A graphical representation of the potential field view is also valuable. There is however a need for better tools and methodologies to aid in the calibration process.

[2] $MHdistance = |x_2 - x_1| + |y_2 - y_1|$

PF based solutions can be less controllable than traditional solutions

In a potential field based solution an often large amount of fields collaborate to form the total potential field which is used by the agents for navigating in the game world. This can lead to interesting emergent behaviors, but can also limit the control over agents compared to pathfinding solutions. A debug tool with graphical representation of the potential fields is of great value to trace what causes possible irrational behavior.

We believe that potential field based solutions can be a successful option to pathfinding algorithms such as A* in many RTS scenarios. The design of subfields and collaboration between several subfields can create interesting and effective emergent behavior, for example to surround enemy units in a half circle to maximize firepower as described in Paper III. It is also easy to create defensive behavior under certain circumstances, for example letting units retreat when their weapon is on cooldown or if they are outnumbered by switching from attracting to repelling fields around opponents units. This is also described in Paper III. In Paper VII we show how potential fields can be combined with pathfinding algorithms to get the best from both worlds.

1.5.2 RTS game architectures

In Paper VII we describe a multi-agent based bot architecture for the very popular, commercial RTS game *StarCraft*. The main goal of the project was to design an architecture where logic and behavior can be modified, changed and added at several levels of abstraction without breaking any core logic of the bot. The main features of the architecture are:

- High-level tasks such as resources planning or base building are handled by global manager agents. It is easy to modify code for a manager, add new managers for specific tasks, or use multiple implementations of one manager using inheritance.
- Combat tasks are divided into three levels; *Commander*, *Squad* and *UnitAgent*. Logic for high-level decisions can be modified in the *Commander* agent, and specialized squad behavior can be added to new *Squad* agents extending the basic *Squad* agent.
- Buildorder, upgradeorder, techorder and squads setup are read from file and can easily be modified or exchanged without any recompilation needed.
- It is possible to create an own agent implementation for each unit or building in the game to micro-manage that specific unit type.

We believe the main goal of the bot architecture is met. The bot supports the flexibility and expandability requirements that were defined.

1.5.3 Gameplay experience in RTS games

In Paper VIII we performed an experiment of how a quite simple runtime difficulty scaling affected some gameplay experience factors such as enjoyment and variety in the ORTS game. The experiment showed slightly higher perceived enjoyment and variety when playing against bots that adapted the difficulty at runtime, however none of the results are statistically significant and much more work has to be done to prove if runtime difficulty scaling in general can enhance gameplay experience in RTS games. Future experiments should preferably be made in a more well-known and popular game such as *StarCraft*.

In Paper IX we tried to identify characteristics of bots and human players in the *Spring* game. In the experiment the participants were asked to watch replays of games from bots facing bots or bots facing humans. The participants were informed that each of the players could either be a bot or a human. The task was to guess and motivate if each player were controlled by a bot or a human player. Although the experiment showed some interesting results more work has to be done preferably in a more well-known game.

1.6 Future Work

Even though we believe the main goal of creating a flexible and epxandable bot architecture is met, there are many possibilities for improvement. Adaptivity is one such improvement. There are several benefits of adaptive AI:

- Changing the bots behavior depending on what the opponent does increase the chance of winning a game.
- Players have difficulty learning a pattern of how a bot plays if it does not always take the same actions under the same circumstances.
- A bot that supports adaptivity can generate interesting emergent behavior.
- An adaptive bot that does not always play in the same way is probably more interesting for human players to play against, thus extending the lifetime of a game.
- A bot that can scale difficulty up or down at runtime can be a challenge for both beginners and expert players.

There are several ways to incorporate adaptivity to the static buildorder/upgrades/techs/squads setup files currently used in the bot.

One way is to have several files for the same player/opponent combination, for example three different buildorder files for Terran vs. Zerg. Which file to use is chosen when the bot is started, either randomly or based on for example map features or win/loss history against a specific opponent. This is not runtime adaptivity, but a simple way of making the bot use different tactics.

Another way is to use a more complex language for the buildorder/upgrades/techs/ squads setup files where rules and conditions can be added, for example have optional squads that only are created if certain conditions are met in the game. This requires a different and more complex language and interpretator. A choice has to be made between using a full scripting language like LUA or create an own specific language. A generic language like LUA would make it very difficult to use for example genetic programming to evolve scripts.

A third way is to split the text files into several parts. The first part can handle the early game where the player has to create basic buildings. The second part can handle the middle game where the player can choose to focus on different units and/or upgrades, and the third part is the end game where the player has access to powerful units and upgrades. Each part can have several versions, and which versions to use can be decided at startup or during runtime. This is illustrated in Figure 1.4. It is important that the parts can be combined, so for example a game does not get stuck in middle game because it requires a building that was not in the early game file. It might be the case that it is not possible to define good generic checkpoints which is required by this solution.

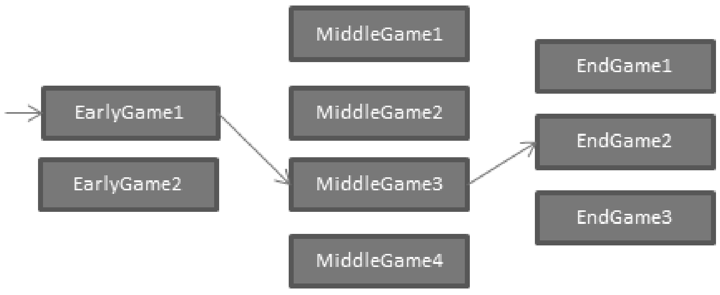


Figure 1.4: *Buildorder/upgrades/techs/squad setup files can be split in parts where each part can have several implementations. The arrow shows the parts choosen for a specific game.*

It is also possible to add adaption to the *Commander* agent. In the current version the *Commander* launch an attack at the enemy once all *Required* squads are filled with units. An interesting feature could be that the *Commander* launch a counter attack if the own base has been attacked and the attackers were repelled. The *Commander* can also use conceptual potential fields, i.e. fields that are not generated by an in game unit or object. Instead they are generated from tactical information about the game world. Areas where the enemy defense is strong can for example generate repelling fields, and areas where an attack can cause severe damage to the enemy can generate attracting fields. Examples of such areas are undefended supply lines where an air attack can quickly kill lots of enemy workers.

Another improvement for the bot is to optimize the buildorder/upgrades/techs/squad setup files. These files are currently hand crafted using our own knowledge of the game. There are lots tips and replays from top players available on numerous fan sites. The idea is to use that information to automatically create effective tactics files. It could also be interesting to use some form of evolutionary system to evolve tactics files.

Regarding gameplay experience there are lots of possible work to be done. One option is to repeat the experiments from Papers VIII and IX in *StarCraft*. Since it is a much more well-known game it is possible that the runtime difficulty scaling and human/bot characteristics experiments will give different results simply because people know how to play *StarCraft*.

We believe that the BTHAI *StarCraft* bot can provide a good basis for future research within RTS game AI.

CHAPTER

TWO

PAPER I

Using Multi-agent Potential Fields in Real-time Strategy Games

Johan Hagelbäck & Stefan J. Johansson

Proceedings of the Seventh International Conference on Autonomous Agents and Multi-agent Systems (AAMAS). 2008.

2.1 Introduction

A *Real-time Strategy (*RTS*)* game is a game in which the players use resource gathering, base building, technological development and unit control in order to defeat its opponent(s), typically in some kind of war setting. The RTS game is not turn-based in contrast to board games such as Risk and Diplomacy. Instead, all decisions by all players have to be made in real-time. Generally the player has a top-down perspective on the battlefield although some 3D RTS games allow different camera angles. The real-time aspect makes the RTS genre suitable for multiplayer games since it allows players to interact with the game independently of each other and does not let them wait for someone else to finish a turn.

Khatib (1986) introduced a new concept while he was looking for a real-time obstacle avoidance approach for manipulators and mobile robots. The technique which he called *Artificial Potential Fields* moves a manipulator in a field of forces. The position to be reached is an attractive pole for the end effector (e.g. a robot) and obstacles are repulsive surfaces for the manipulator parts. Later on Arkin (1987) updated the knowledge by creating another technique using superposition of spatial vector fields in order to generate behaviours in his so called motor schema concept.

Many studies concerning potential fields are related to spatial navigation and obstacle avoidance, see e.g. Borenstein and Koren (1991); Khatib (2004); Massari et al. (2004). The technique is really helpful for the avoidance of simple obstacles even though they are numerous. Combined with an autonomous navigation approach, the result is even better, being able to surpass highly complicated obstacles (Borenstein & Koren, 1989). However most of the premises of these approaches are only based on repulsive potential fields of the obstacles and an attractive potential in some goal for the robot (Vadakkepat, Tan, & Ming-Liang, 2000).

Lately some other interesting applications for potential fields have been presented. The use of potential fields in architectures of multi agent systems is giving quite good results defining the way of how the agents interact. Howard, Matarić, and Sukhatme (2002) developed a mobile sensor network deployment using potential fields, and potential fields have been used in robot soccer (Johansson & Saffiotti, 2002; Röfer et al., 2004). Thurau et al. (2004b) has developed a game bot which learns reactive behaviours (or potential fields) for actions in the First-Person Shooter (FPS) game Quake II through imitation.

In some respect, videogames are perfect test platforms for multi-agent systems. The environment may be competitive (or even hostile) as in the case of a FPS game. The NPCs (e.g. the units of the opponent army in a war strategy game) are supposed to act rationally and autonomously, and the units act in an environment which enables explicit communication and collaboration in order to be able to solve certain tasks.

Previous work on describing how intelligent agent technology has been used in videogames include the extensive survey of Niederberger and Gross (2003) and early work by vanLent et al. (1999). Multi-agent systems has been used in board games by Kraus and Lehmann (1995) who addressed the use of MAS in Diplomacy and Johansson (2006) who proposed a general MAS architecture for board games.

The main research question of this paper is: *Is Multi-agent Potential Fields (*MAPF*) an appropriate approach to implement highly configurable bots for* RTS *games?* This breaks down to:

1. How does MAPF perform compared to traditional solutions?

2. To what degree is MAPF an approach that is configurable with respect to variations in the domain?

We will use a proof of concept as our main methodology where we compare an implementation of MAPF playing ORTS with other approaches to the game. The comparisons are based both on practical performance in the yearly ORTS tournament, and some theoretical comparisons based on the descriptions of the other solutions.

First we describe the methodology that we propose to follow for the design of a MAPF bot. In Section 2.3 we describe the test environment. The creation of our MAPF player follows the proposed methodology and we report on that in Section 2.4. The experiments and their results are described in Section 2.5. We finish off by discussing, drawing some conclusions and outlining future work in Sections 2.6–2.7.

2.2 A Methodology for Multi-agent Potential Fields

When constructing a multi-agent system of potential field controlled agents in a certain domain, there are a number of issues that have to be dealt with. To structure this, we identify six phases in the design of a MAPF-based solution:

1. The identification of objects,
2. The identification of the driving forces (fields) of the game,
3. The process of assigning charges to the objects,
4. The granularity of time and space in the environment,
5. The agents of the system, and
6. The architecture of the MAS.

In *the first phase*, we may ask us the following questions: What are the *static objects* of the environment? That is: what objects remain their attributes throughout the lifetime of the scenario? What are the *dynamic objects* of the environment? Here we may identify a number of different ways that objects may change. They may move around, if the environment has a notion of physical space. They may change their attractive (or repulsive) impact on the agents. What are the *modifiability* of the objects? Some objects may be consumed, created, or changed by the agents.

In *the second phase*, we identify the driving forces of the game at a rather abstract level, e.g. to avoid obstacles, or to base the movements on what the opponent does. This leads us to a number of fields. The main reason to enable multiple fields is that it is very easy to isolate certain aspects of the computation of the potentials if we are able to filter out a certain aspect of the overall potential, e.g. the repulsive forces generated by the terrain in a physical environment. We may also dynamically weight fields separately, e.g. in order to decrease the importance of the navigation field when a robot stands still in a surveillance mission (and only moves its camera). We may also have *strategic fields* telling the agents in what direction their next goal is, or *tactical fields* coordinating the movements with those of the team-mate agents.

The third phase include to place the objects in the different fields. Static objects should perhaps be in the *field of navigation*. Typically, the potentials of such a field is pre-calculated in order to save precious run time CPU resources.

In *the fourth phase*, we have to decide the resolution of space and time. If the agents are able to move around in the environment, both these measures have an impact on the look-ahead. The space resolution, since it decides where in space we are able to go, and the time in that it determines how far we may get in one time frame.

The fifth phase, is to decide what objects to agentify and set the repertoire of those agents: what actions are we going to evaluate in the look-ahead? As an example, if the agent is omnidirectional in its movements, we may not want to evaluate all possible points that the agent may move to, but rather try to filter out the most promising ones by using some heuristic, or use some representable sample.

In *the sixth step*, we design the architecture of the MAS. Here we take the unit agents identified in the fifth phase, give them roles and add the supplementary agents (possibly) needed for coordination, and special missions (not covered by the unit agents).

2.3 ORTS

Open Real Time Strategy (ORTS) (Buro, 2007a) is a real-time strategy game engine developed as a tool for researchers within artificial intelligence (AI) in general and game AI in particular. ORTS uses a client-server architecture with a game server and players connected as clients. Each timeframe clients receive a data structure from the server containing the current game state. Clients can then issue commands for their units. Commands can be like move unit A to (x, y) or attack opponent unit X with unit A. All client commands are executed in random order by the server.

Users can define different type of games in scripts where units, structures and their interactions are described. All type of games from resource gathering to full real time strategy (RTS) games are supported. We focus on two types of two-player games, *tankbattle* and *tactical combat*. These games were part of the 2007 years ORTS competition (Buro, 2007a).

- In *Tankbattle* each player has 50 tanks and five bases. The goal is to destroy the bases of the opponent. Tanks are heavy units with long fire range and devastating firepower but a long cool-down period, i.e. the time after an attack before the unit is ready to attack again. Bases can take a lot of damage before they are destroyed, but they have no defence mechanism of their own so it may be important to defend own bases with tanks. The map in a tankbattle game has randomly generated terrain with passable lowland and impassable cliffs.

- In *Tactical combat* each player has 50 marines and the goal is to destroy all the marines of the opponent. Marines have short fire range, average firepower and a short indestructible period. They are at the start of the game positioned randomly at either right or left side of the map. The map does not have any impassable cliffs.

Both games contain a number of neutral units (sheep). These are small and (for some strange reason) indestructible units moving randomly around the map. The purpose of sheep are to make pathfinding and collision detection more complex.

2.4 MAPF in ORTS

We have implemented an ORTS client for playing both Tankbattle and Tactical Combat based on MAPF following the proposed methodology. Below we will describe the creation of our MAPF solution.

2.4.1 Identifying objects

We identify the following objects in our applications: Cliffs, Sheep, and own (and opponent) tanks, marines and base stations.

2.4.2 Identifying fields

We identified four driving forces in ORTS: Avoid colliding with moving objects, Hunt down the enemy's forces and for the Tankbattle game also to Avoid colliding with cliffs, and to Defend the bases. This leads us to three types of potential fields: *Field of Navigation*, *Strategic Field*, and *Tactical field*.

The field of navigation is generated by repelling static terrain. We would like agents to avoid getting too close to objects where they may get stuck, but instead smoothly pass around them.

The strategic field is an attracting field. It makes agents go towards the opponents and place themselves on an appropriate distance where they can fight the enemies.

Own units, own bases and sheep generate small repelling fields. The purpose is that we would like our agents to avoid colliding with each other or bases as well as avoiding the sheep.

2.4.3 Assigning charges

Each unit (own or enemy), control center, sheep and cliffs has a set of charges which generates a potential field around the object. Below you will find a more detailed description of the different fields. All fields generated by objects are weighted and summed to form a total field which is used by agents when selecting actions. The actual formulas for calculating the potentials very much depend on the application.

Figure 3.2 in Paper II shows a 2D view of the map during a tankbattle game. It shows our agents (green) moving in to attack enemy bases and units (red). Figure 3.3 shows the potential field view of the same tankbattle game. Dark areas has low potential and light areas high potential. The light ring around enemy bases and units, located at maximum shooting distance of our tanks, is the distance our agents prefer to attack opponent units from. It is the final move goal for our units.

Cliffs Cliffs generate a repelling field for obstacle avoidance. The potential $p_{cliff}(d)$ at distance d (in tiles) from a cliff is:

$$p_{cliff}(d) = \begin{cases} -80/d^2 & \text{if } d > 0 \\ -80 & \text{if } d = 0 \end{cases} \tag{2.1}$$

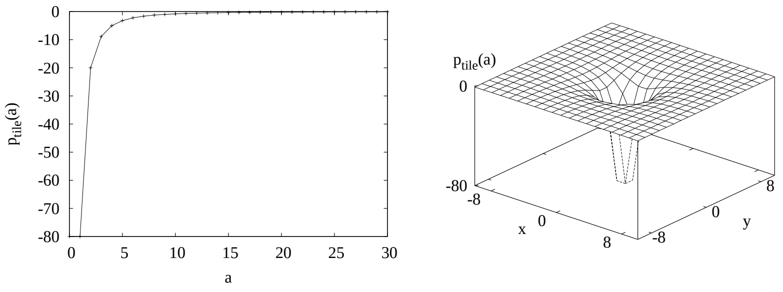


Figure 2.1: *The potential $p_{cliff}(d)$ generated by a cliff given the distance d.*

Note that if more than one cliff affects the same potential field tile, the actual potential is not calculated as the sum of the potentials (as in the other fields) but rather as the lowest value. This approach works better for passages between cliffs, see Figure 2.1.

The navigation field is post-processed in two steps to improve the agents abilities to move in narrow passages and avoid dead ends. The first step is to fill dead ends. The pseudo code in Figure 2.2 describes how this is done.

```
**for all** $x, y$ in navigation field $F(x, y)$ **do**
   **if** is_passable($x, y$) **then**
      $blocked = 0$
      **for all** 16 directions around $x, y$ **do**
         **if** cliff_within(5) **then**
            $blocked = blocked + 1$
         **end if**
      **end for**
      **if** $blocked >= 9$ **then**
         $IMPASSABLE(x, y) = true$
      **end if**
   **end if**
**end for**
```

Figure 2.2: *Pseudo code for filling dead ends.*

For each passable tile (x, y) , we check if there are cliffs within 5 tiles in all 16 directions. If 9 or more directions are blocked by cliffs, we consider tile (x, y) impassable (Figure 2.3).

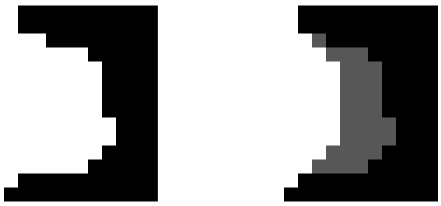


Figure 2.3: *Example of the navigation field before and after filling dead ends. White are passable tiles, black impassable tiles and grey tiles filled by the algorithm.*

Next step is to clear narrow passages between cliffs from having a negative potential. This will make it easier for agents to use the passages, see Figure 2.5. Figure 2.4 shows the pseudo code for this processing step. For each passable tile (x, y) with negative potential, check if adjacent tiles has even lower negative potentials. If so, (x, y) is probably in a narrow passage and its potential is set to 0.

```
for all $x, y$ in navigation field $F(x, y)$ do
  $potential = p(x, y)$
  if $potential >= -50\ AND\ potential <= -1$ then
    if $p(x-1, y) < potential\ AND\ p(x+1, y) < potential$
    then
      $p(x, y) = 0$
    end if
    if $p(x, y-1) < potential\ AND\ p(x, y+1) < potential$
    then
      $p(x, y) = 0$
    end if
  end if
end for
```

Figure 2.4: *Pseudo code for clearing narrow passages.*

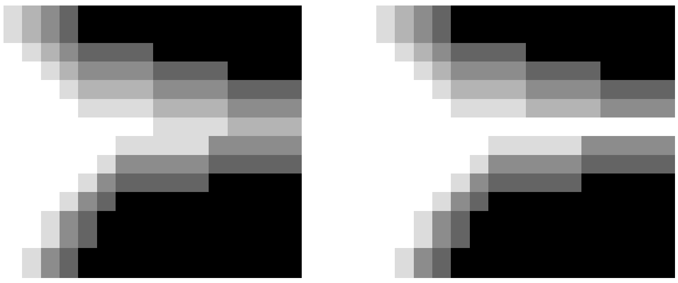


Figure 2.5: *Example of the navigation field before and after clearing passages. White tiles has potential 0, and the darker the colour the more negative potential a tile has.*

The opponent units All opponent units generates a symmetric surrounding field where the highest potential is in a ring around the object with a radius of MSD (*Maximum Shooting Distance*). As illustrated in Figure 2.6, MDR refers to the *Maximum Detection Range*, the distance from which an agent starts to detect the opponent unit. In general terms, the $p(d)$-function can be described as:

$$p(d) = \begin{cases} k_1 d, & \text{if } a \in [0, MSD - a[\\ c_1 - d, & \text{if } a \in [MSD - a, MSD] \\ c_2 - k_2 d, & \text{if } a \in]MSD, MDR] \end{cases} \tag{2.2}$$

Unit	k_1	k_2	c_1	c_2	MSD	a	MDR
Marine	2	0.15	24.5	15	7	2	100
Tank	2	0.22	24.1	15	7	2	68
Base	3	0.255	49.1	15	12	2	130

Table 2.1: *The parameters used for the generic $p(d)$-function of Eq. 2.2.*

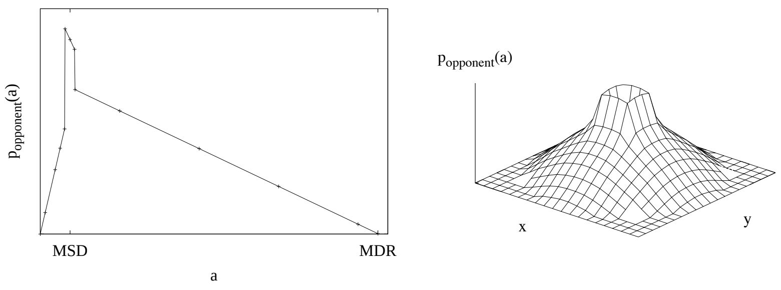


Figure 2.6: *The potential $p_{opponent}(d)$ generated by the general opponent function given the distance d.*

Own bases Own bases generate a repelling field for obstacle avoidance. Below is the function for calculating the potential $p_{ownbase}(d)$ at distance d (in tiles) from the center of the base. Note that 4 is half the width of the base, and distances less than or equal to this value has a much lower potential. This approximation is not entirely correct at the corners of the base (since the base is quadratic rather than circular, see Figure 2.7), but it works well in practice.

$$p_{ownbase}(d) = \begin{cases} 5.25 \cdot d - 37.5 & \text{if } d <= 4 \\ 3.5 \cdot d - 25 & \text{if } d \in]4, 7.14] \\ 0 & \text{if } d > 7.14 \end{cases} \tag{2.3}$$

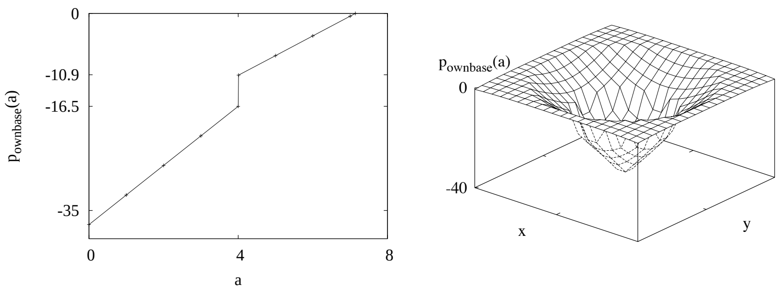


Figure 2.7: *The repelling potential $p_{ownbase}(d)$ generated by the own bases given the distance d.*

The own mobile units — tanks and marines Own units, agents, generate a repelling field for obstacle avoidance (see Figure 2.8). In general terms, the potential $p_{ownunit}(d)$ at distance d (in tiles) from the center of an agent is calculated as:

$$p_{ownunit}(d) = \begin{cases} -20 & \text{if } d <= radius \\ d \cdot k - c & \text{if } d \in]radius, l], \\ 0 & \text{if } d >= l \end{cases} \tag{2.4}$$

Unit	$radius$	k	c	l
Marine	0.5	3.2	5.6	1.75
Tank	0.875	3.2	10.8	3.375

Table 2.2: *The parameters used for the generic $p_{ownunit}(d)$-function of Eq. 2.4.*

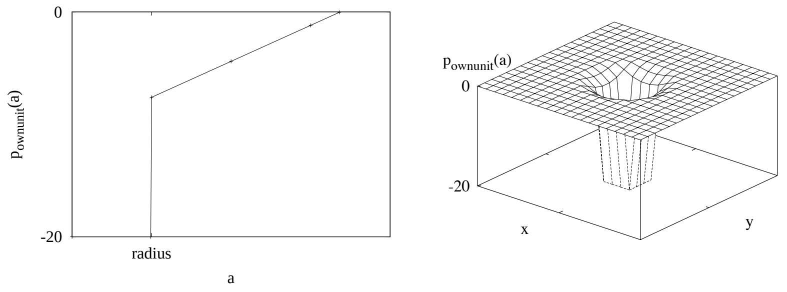


Figure 2.8: *The repelling potential $p_{ownunit}(d)$ generated by the generic function given the distance d.*

Sheep Sheep generate a small repelling field for obstacle avoidance. The potential $p_{sheep}(d)$ (depicted in Figure 2.9) at distance d (in tiles) from the center of a sheep is calculated as:

$$p_{sheep}(d) = \begin{cases} -10 & \text{if } d <= 1 \\ -1 & \text{if } d \in]1, 2] \\ 0 & \text{if } d > 2 \end{cases} \quad (2.5)$$

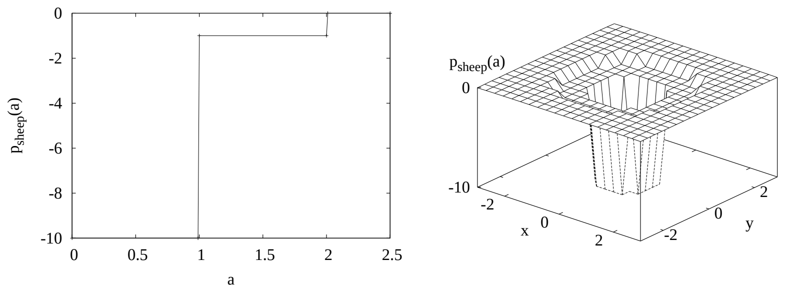


Figure 2.9: *The potential $p_{sheep}(d)$ generated by a sheep given the distance d.*

2.4.4 On the granularity

When designing the client we had to decide a resolution for the potential field. A tank-battle game has a map of 1024x1024 points and the terrain is constructed from tiles of

16x16 points. After some initial tests we decided to use 8x8 points for each tile in the potential field. The resolution had to be detailed enough for agents to be able to move around the game world using only the total potential field, but a more detailed resolution would have required more memory and the different fields would have been slower to update.[1] Thus in our implementation 8x8 points was found to be a good trade-off.

2.4.5 The unit agent(s)

When deciding actions for an agent, the potential of the tile the agent is at is compared with the potentials of the surrounding tiles. The agent moves to the center of the neighbour tile with the highest potential, or is idle if the current tile is highest. If an agent has been idle for some time, it moves some distance in a random direction to avoid getting stuck in a local maxima. If an opponent unit is within fire range, the agent stops to attack the enemy.

Since there is an advantage of keeping the agents close to the maximum shooting distance (MSD), the positions of the opponent units are not the final goal of navigation. Instead we would like to keep them near the MSD. The obstacles should be avoided, roughly in the sense that the further away they are, the better it is. Here, the own agents are considered to be obstacles (for the ability to move).

When an agent executes a move action, the tactical field is updated with a negative potential (same as the potential around own agents) at the agents destination. This prevents other agents from moving to the same position if there are other routes available.

2.4.6 The MAS architecture

In a tank-battle game our agents has two high-level tactical goals. If we have a numerical advantage over the opponent units we attack both bases and units. If not, we attack units only and wait with attacking bases. For agents to attack both units and bases, one of the following constraints must be fulfilled:

- We must have at least twice as many tanks as the opponent
- The opponent have less than six tanks left
- The opponent have only one base left

If none of these constraints are fulfilled, the tactical goal is to attack opponent units only. In this case the field generated by opponent bases are not an attracting field. Instead they generate a repelling field for obstacle avoidance (same as the field generated by own bases). We want to prevent our agents from colliding with opponent bases if their goal is not to attack them. In a tactical combat game no bases are present and agents always aim to destroy opponent marines.

[1]The number of positions quadruples as the resolution doubles.

Attack coordination

We use a coordinator agent to globally optimise attacks at opponent units. The coordinator aims to destroy as many opponent units as possible each frame by concentrating fire on already damaged units. Below is a description of how the coordinator agent works. After the coordinator is finished we have a near-optimal allocation of which of our agents that are dedicated to attack which opponent units or bases.

The coordinator uses an attack possibility matrix. The $i \times k$ matrix A defines the opponent units i (out of n) within MSD which can be attacked by our agents k (out of m) as follows:

$$a_{k,i} = \begin{cases} 1 & \text{if the agent } k \text{ can attack opponent unit } i \\ 0 & \text{if the agent } k \text{ cannot attack opponent unit } i \end{cases} \tag{2.6}$$

$$A = \begin{bmatrix} a_{0,0} & \cdots & a_{m-1,0} \\ \vdots & \ddots & \vdots \\ a_{0,n-1} & \cdots & a_{m-1,n-1} \end{bmatrix} \tag{2.7}$$

We also need to keep track of current hit points (HP) of the opponent units i as:

$$HP = \begin{bmatrix} HP_0 \\ \vdots \\ HP_{n-1} \end{bmatrix} \tag{2.8}$$

Let us follow the example below to see how the coordination heuristic works.

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad HP = \begin{bmatrix} HP_0 = 2 \\ HP_1 = 3 \\ HP_2 = 3 \\ HP_3 = 4 \\ HP_4 = 4 \\ HP_5 = 3 \end{bmatrix} \tag{2.9}$$

First we sort the rows so the highest priority targets (units with low HP) are in the top rows. This is how the example matrix looks like after sorting:

$$A_2 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \quad HP = \begin{bmatrix} HP_0 = 2 \\ HP_1 = 3 \\ HP_2 = 3 \\ HP_5 = 3 \\ HP_4 = 4 \\ HP_3 = 4 \end{bmatrix} \tag{2.10}$$

Next step is to find opponent units that can be destroyed this frame (i.e. we have enough agents able to attack an opponent unit to reduce its HP to 0). In the example we have enough agents within range to destroy unit 0 and 1. We must also make sure that the agents attacking unit 0 or 1 are not attacking other opponent units in A. This is done by assigning a 0 value to the rest of the column in A for all agents attacking unit 0 or 1.

Below is the updated example matrix. Note that we have left out some elements for clarity. These has not been altered in this step and are the same as in matrix A_2.

$$A_3 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & & & 0 & \\ 0 & 0 & 0 & 0 & & & 0 & \\ 0 & 0 & 0 & 0 & & & 0 & \\ 0 & 0 & 0 & 0 & & & 0 & \end{bmatrix} \quad HP = \begin{bmatrix} HP_0 = 2 \\ HP_1 = 3 \\ HP_2 = 3 \\ HP_5 = 3 \\ HP_4 = 4 \\ HP_3 = 4 \end{bmatrix} \tag{2.11}$$

The final step is to make sure the agents in the remaining rows (3 to 6) only attacks one opponent unit each. This is done by, as in the previous step, selecting a target i for each agent (start with row 3 and process each row in ascending order) and assign a 0 to the rest of the column in A for the agent attacking i. This is how the example matrix looks like after the coordinator is finished:

$$A_4 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad HP = \begin{bmatrix} HP_0 = 2 \\ HP_1 = 3 \\ HP_2 = 3 \\ HP_5 = 3 \\ HP_4 = 4 \\ HP_3 = 4 \end{bmatrix} \tag{2.12}$$

In the example the fire coordinator agent have optimised attacks to:

- Unit 0 is attacked by agents 0 and 3. It should be destroyed.
- Unit 1 is attacked by agents 1, 2 and 6. It should be destroyed.
- Unit 5 is attacked by agent 6. Its HP should be reduced to 2.
- Unit 4 is attacked by agents 4 and 5. Its HP should be reduced to 2.
- Units 2 and 3 are not attacked by any agent.

The Internals of the Coordinator Agent

The coordinator agent first receive information from each of the own agents. It contains its positions and ready-status, as well as a list of the opponent units that are within range. Ready-status means that an agent is ready to fire at enemies. After an attack a unit has a cool-down period while it cannot fire. From the server, it will get the current hit point status of the opponent units.

Now, the coordinator filters the agent information so that only those agents that are i) ready to fire and ii). have at least one opponent unit within MSD, are left.

For each agent k that is ready to fire, we iterate through all opponent units and bases. To see if k can attack unit i we use a three level check:

1. Agent k must be within Manhattan distance[2] * 2 of i (very fast but inaccurate calculation)

2. Agent k must be within real (Euclidean) distance of i (slower but accurate calculation)

3. Opponent unit i must be in line of sight of k (very slow but necessary to detect obstacles in front of i)

The motivation behind the three-level check is to start with fast but inaccurate calculations, and for each level passed a slower and more accurate check is performed. This reduces CPU usage by skipping demanding calculations such as line-of-sight for opponent units or bases that are far away.

Next step is to sort the rows in A in ascending order based on their HP (prioritise attacking damaged units). If two opponent units has same hit points left, the unit i which can be attacked by the largest number of agents k should be first (i.e. concentrate fire to damage a single unit as much as possible rather than spreading the fire). When an agent attacks an opponent unit it deals a damage value randomly chosen between the attacking unit's minimum (min_{dmg}) and maximum (max_{dmg}) damage. A unit hit by an attack get its HP reduced by the damage value of the attacking unit minus its own armour value. The armour value is static and a unit's armour cannot be destroyed.

The next step is to find opponent units which can be destroyed this frame. For every opponent unit i in A, check if enough agents u can attack i to destroy it as:

$$(\sum_{k=0}^{m-1} a(k,i)) \cdot (damage_u - armour_i) >= HP_i \tag{2.13}$$

$armour_i$ is the armour value for the unit type of i (0 for marines and bases, 1 for tanks) and $damage_u = min_{dmg} + p \cdot (max_{dmg} - min_{dmg})$, where $p \in [0, 1]$. We have used a p value of 0.75, but it can be changed to alter the possibility of actually destroying opponent units.

If more agents can attack i than is necessary to destroy it, remove the agents with the most occurrences in A from attacking i. The motivation behind this is that the agents u with most occurrences in A has more options when attacking other units.

[2]The Manhattan distance between two coordinates (x_1, y_1), (x_2, y_2) is given by $abs(x_1 - x_2) + abs(y_1 - y_2)$.

At last we must make sure the agents attacking i does not attack other opponent units in A. This is done by assigning a 0 value to the rest of the column.

The final step is to make sure agents not processed in the previous step only attacks one opponent unit each. Iterate through every i that cannot be destroyed but can be attacked by at least one agent k, and assign a 0 value to the rest of the column for each k attacking i.

2.5 Experiments

Our bot have participated in the 2007 years ORTS competition. Below is a brief description of the other competition entries (Buro, 2007a). The results from the competition are presented in Tables 2.3–2.4. As we can see from the results summary our bot was not among the top entries in the competition, but rather in the bottom half. We did however win almost a third of the played games in both categories. Note that all other competition entries are based on more traditional approaches with pathfinding and higher level planning, and our goal is to investigate if our Multi-agent Potential Fields based bot is able to reach the same level of performance as the traditional solutions.

Team	Wins ratio	Wins/games	Team name
nus	98%	(315/320)	National Univ. of Singapore
WarsawB	78%	(251/320)	Warsaw Univ., Poland
ubc	75%	(241/320)	Univ. of British Columbia, Canada
uofa	64%	(205/320)	Univ. of Alberta, Canada
uofa.06	46%	(148/320)	Univ. of Alberta
BTH	**32%**	**(102.5/320)**	**Blekinge Inst. of Tech., Sweden**
WarsawA	30%	(98.5/320)	Warsaw University, Poland
umaas.06	18%	(59/320)	Univ. of Maastricht, The Netherlands
umich	6%	(20/320)	Univ. of Michigan, USA

Table 2.3: *Summary of the results of ORTS tank-battle 2007*

2.5.1 Opponent Descriptions

The team *NUS* use finite state machines and influence maps in high-order planning on group level. The units in a squad spread out on a line and surround the opponent units at MSD. Units use the cool-down period to keep out of MSD. Pathfinding and a flocking algorithm is used to avoid collisions.

UBC gather units in squads of 10 tanks or marines. Squads can be merged with other squads or split into two during the game. Pathfinding is combined with force fields to

Team	Wins ratio	Wins/games	Team name
nus	99%	(693/700)	National Univ. of Singapore
ubc	75%	(525/700)	Univ. of British Columbia, Canada
WarsawB	64%	(451/700)	Warsaw Univ., Poland
WarsawA	63%	(443/700)	Warsaw Univ., Poland
uofa	55%	(386/700)	Univ. of Alberta, Canada
BTH	**28%**	**(198/700)**	**Blekinge Inst. of Tech., Sweden**
nps	15%	(102/700)	Naval Postgraduate School, USA
umich	0%	(2/700)	Univ. of Michigan, USA

Table 2.4: *Summary of the results of the ORTS tactical combat*

avoid obstacles and bit-mask for collision avoidance. Units spread out at MSD when attacking. Weaker squads are assigned to weak spots or corners of the opponent unit cluster. If an own base is attacked, it may decide to try to defend the base.

WarsawA synchronises units by assigning each unit position to a node in a grid. The grid is also used for pathfinding. When units are synchronised they attack the enemy at a line going for its weakest spots at a predefined distance.

WarsawB uses pathfinding with an additional dynamic graph for moving objects. Own units uses a repelling force field collision avoidance. Units are gathered in one large squad. When the squad attacks, its units spread out on a line at MSD and each unit attack the weakest opponent unit in range. In tactical combat, each own unit is assigned to an opponent unit and it always tries to be at the same horizontal line (y coordinate) as its assigned unit.

Uofa uses a hierarchical commander approach ranging from squad commanders down to pathfinding and attack coordination commanders. Units are grouped in a single, large cluster and tries to surround the opponent units by spreading out at MSD. The hierarchical commander approach is not used in tactical combat.

Umich uses an approach where the overall tactics are implemented in the SOAR language. SOAR in turn have access to low-level finite state machines for handling, for example, squad movement. Units are gathered in a single squad hunting enemies, and opponent units attacking own bases are the primary goals.

Umaas and *Uofa* entered the competition with their 2006 years entries. No entry descriptions are available.

2.6 Discussion

We discuss potential fields in general, then the results of the experiments, and finally write a few words about the methodology.

2.6.1 The use of PF in games

Traditionally the use of potential fields (PF), although having gained some success in the area of robotic navigation, has been limited in the domain of game AI. There are a number of more or less good reasons for that:

1. PF are considered to be less controllable than traditional planning (Tomlinson, 2004). This may be an important feature in the early stages of a game development.

2. A* and different domain specific improvements of it has proven to gain sufficiently good results.

3. PF based methods are believed to be hard to implement and to debug. These problems may especially apply to the representation of the environment, and the dynamic stability (Tomlinson, 2004).

4. Agents navigating using PFs often get stuck in local optima.

However, from the reported use of potential fields in the area of RoboCup and games indicate that:

PF may be implemented in a way that use the processing time efficiently, especially in highly dynamic environments where lots of objects are moving and long term planning is intractable. By just focusing on nine options (eight directions + standing still) we do, at most, have to calculate the potentials of $9n$ positions for our n units. All potential functions may be pre-calculated and stored in arrays, which makes the actual calculation of the potential of a position just a matter of summing up a number of array elements.

By using multiple maps over the potential landscape (e.g. one for each type of unit), the debug process becomes significantly more efficient. We used different potential landscapes that were put on the map to illustrate the potentials using different colours.

The great thing with PFs is that the attracting – repelling paradigm is very intuitive: the good outcomes of actions are attractive, and the bad outcomes repellent. Thus an action that lead to both bad and good outcomes can be tuned at the outcome level, rather than on the action level.

In static environments, the local optima problem has to be dealt with when using PF. In ORTS, which in some cases is surprisingly static, we used convex filling and path clearing of the terrain to help the units, but this did not always help. We believe that more efforts here will improve the performance. Thurau et al. (2004b) describes a solution to the local maxima problem called avoid-past potential field forces. Each of their agents generate a trail of negative potential, similar to a pheromone trail used by ants, at visited positions. The trail pushes the agent forward if it reaches a local maximum. This approach may work for our agents as well.

2.6.2 The Experiments

There are a number of possible explanations for the good results of the top teams (and the comparative bad results for our team). First, the top teams are very good at handling difficult terrain which, since the terrain is generated randomly, sometimes cause problems for our agents due to local optima.

The second advantage is coordinating units in well-formed squads. Since we do not have any attracting mechanism between agents and higher-level grouping of squads, our agents are often spread out with a large distance between them. Enemies can in some cases destroy our agents one at a time without risk of being attacked by a large number of coordinated agents.

The third advantage is that the top teams spread out units at MSD, and always tries to keep that distance. Since the field of opponents are a sum of the generated potentials for all opponent units, the maxima tend to be in the center of the opponent cluster and our agents therefore attack the enemy at their strongest locations instead of surrounding the enemy.

We believe it is possible to solve these issues using MAPF. The first issue is a matter of details in the resolution of the MAPF. Our agents move to the center of the 8x8 points tile with highest potential. This does not work very well for narrow passages or if bases, other agents or sheep are close. This could be solved by either increase the resolution of the MAPF or add functionality for estimating a potential at a point to enable movement at point level.

The second issue can be solved by using a both positive and negative field for agents. Close to the agents, there is a surrounding negative field as in our implementation, which in turn is surrounded by a positive one. The positive field will make the agents to keep an appropriate distance and possibly having an emergent effect of surrounding the opponent (see e.g. Mamei and Zambonelli (2004)).

The third issue can be solved by not calculating the potential in a point as the sum of the potentials all opponent units generate in that point, but rather the highest potential an opponent unit generate in the point. This will make sure the maxima in the strategic field always are at MSD even if the opponent units are clustered in large groups, and our agents will more likely surround the enemy.

To further improve our bot a new type of tactics field can be used. By generating a large positive field at the weakest spot of the opponent units cluster, agents attack the weakest spot instead of attacking strong locations. This field differs from the other fields used in that it is not generated by a game object, but rather generated by a higher-level tactical decision.

2.6.3 On the Methodology

We chose to implement and test our idea of using a Multi-agent Potential Field based solution in the yearly ORTS competition. As a testbed, we believe that it is good for this purpose for a number of reasons: i). It is a competition, meaning that others will do their best to beat us. ii) It provides a standardised way of benchmarking Game AI solutions iii). The environment is open source and all of the mechanics are transparent. iv) ORTS uses a client-server architecture where clients only has access to the information sent by the server. No client can gain an advantage by hacking the game engine as often is possible in a peer-to-peer architecture. v) Even though ORTS is written in C++ the communication protocol is public and it is possible to write a wrapper to any other language. The results may seem modest, but we show that MAPFs is an alternative to A* based solutions in the case of ORTS. We have no reason to believe that MAPF would not be successful in other RTS games.

2.7 Conclusions and Future Work

A long-term plan, for example path finding, generated by an agent might need re-planning if the game world changes during the execution of the plan. With a PF based solution path planning may be replaced by one step look-ahead, if the analysis is carried out carefully, but yet efficiently. We believe that in ORTS, MAPFs fulfils the requirements of efficiency and flexibility and conclude that MAPF is indeed an interesting alternative worth investigating further. However, more research is needed on how to implement MAPF based solutions in general, and on what tools to use in the debugging and calibration process. Preliminary late results show that our MAPF solution now beat all the competitors of the 2007 ORTS competition. The future of MAPF looks bright and we hope to be able to report further on this in the near future. Future work include to optimise the parameters using e.g. genetic algorithms, to take care of the issues mentioned in Section 2.6, and to refine the agent perspective through distributing the coordination of attacks and the exploration of the map explicitly. We would also like to try our approach in other domains.

CHAPTER
THREE

PAPER II

Demonstration of Multi-agent Potential Fields in Real-time Strategy Games

Johan Hagelbäck & Stefan J. Johansson
Demo Paper on the Seventh International Conference on Autonomous Agents and Multi-agent Systems (AAMAS). 2008.

3.1 The ORTS environment

Open Real Time Strategy (ORTS) (Buro, 2007a) is a real-time strategy game engine developed as a tool for researchers within artificial intelligence (AI) in general and game AI in particular, see Figure 3.1. ORTS uses a client-server architecture with a game server and players connected as clients. Each timeframe clients receive a data structure from the server containing the current game state. Clients can then issue commands for their units. Commands can be like move unit A to (x, y) or attack opponent unit X with unit A. All client commands are executed in random order by the server.

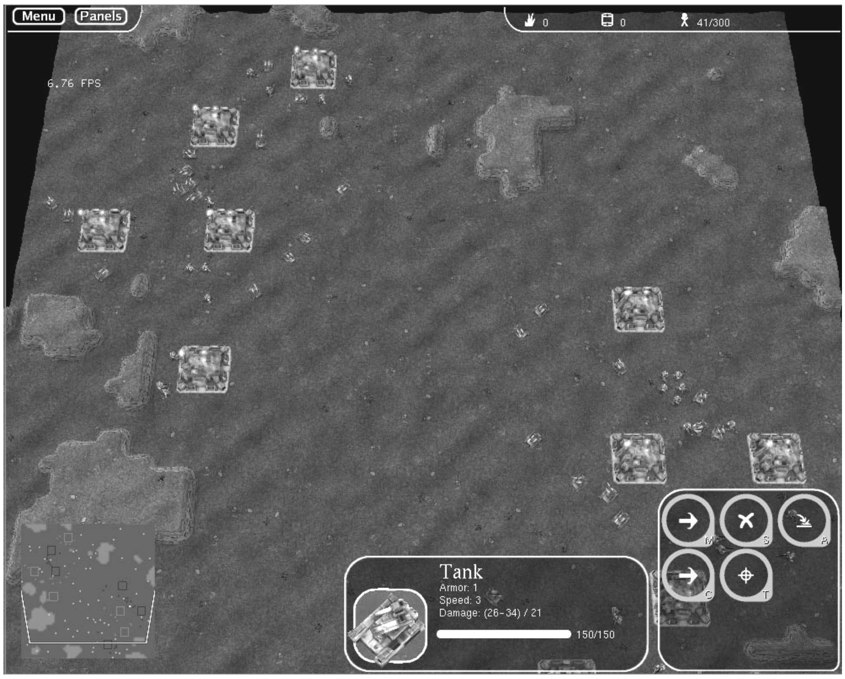


Figure 3.1: *The 3D view of the ORTS Tankbattle game.*

3.2 The used technology

Khatib (1986) introduced a new concept while he was looking for a real-time obstacle avoidance approach for manipulators and mobile robots. The technique which he called *Artificial Potential Fields* moves a manipulator in a field of forces. The position to be reached is an attractive pole for the end effector (e.g. a robot) and obstacles are repulsive surfaces for the manipulator.

Although being a well-known technology in robotics, potential fields has not gained very much interest in the game industry. We show that, not only is it an efficient and robust solution for navigation of a single unit, it is also an approach that works very well in distributed settings of multiple agents. Figure 3.3 shows the potential fields for the green team.

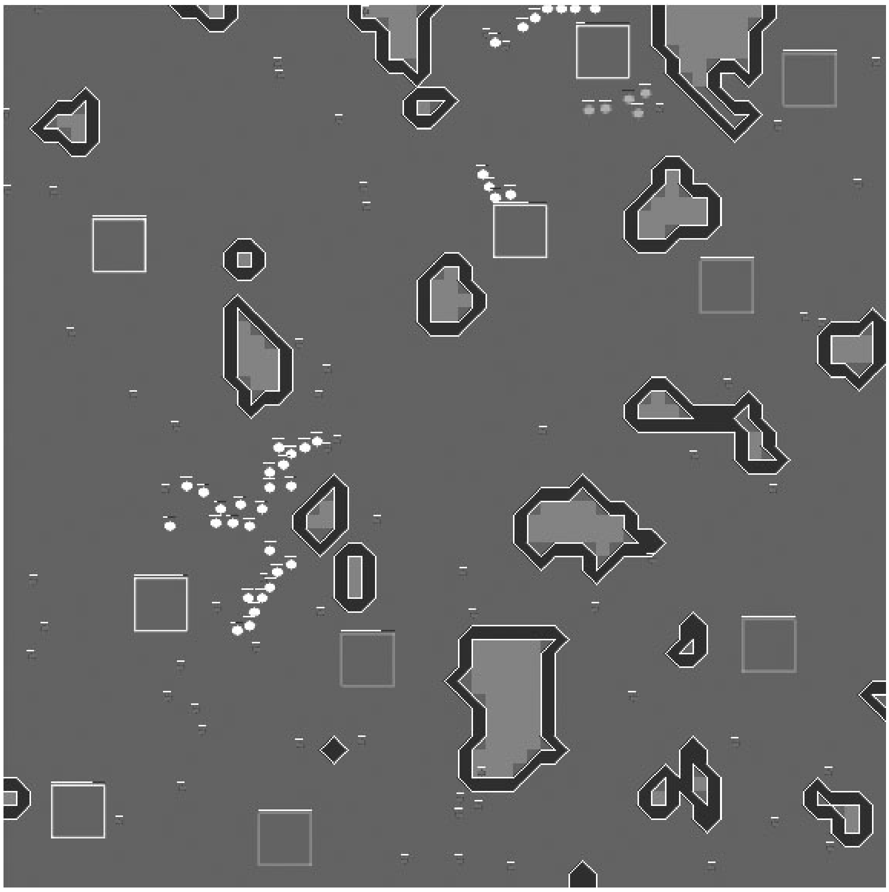


Figure 3.2: *The 2D view of the same ORTS Tankbattle game.*

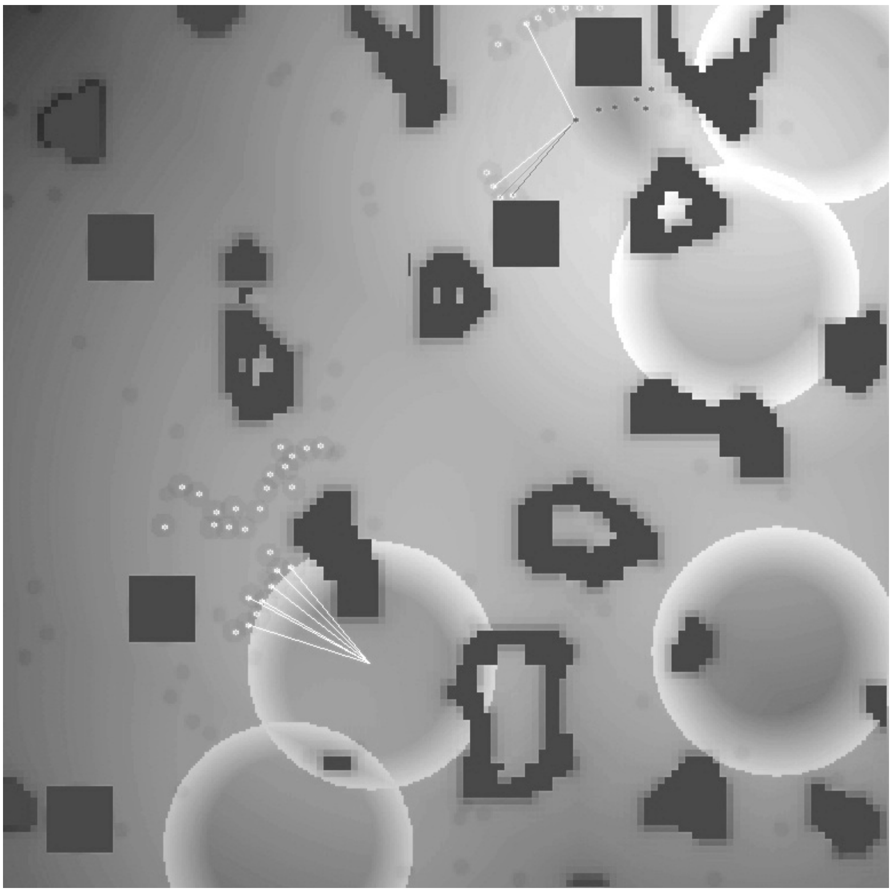


Figure 3.3: *The potential field generated by the units and the terrain. The white lines illustrate the coordinated attacks on a base (lower left) and a unit (upper right).*

3.3 The involved multi-agent techniques

There are several issues to be addressed in an RTS game. First, all units are moving in parallel, which means that they will have to coordinate their movement in some way without bumping into each other, or the surrounding environment. We use potential fields similar to the ones used by e.g. Mamei and Zambonelli (2004) to let the units keep themselves at the right distance.

Second, to improve the efficiency, we coordinate their attacks through the use of a central military commander. This agent is not embodied in the field, but makes sure that no extra shots are spent on opponent units that are already under lethal attack. This is important, since there is a cool-down period during which the units can not attack after a shot.

Third, the commander chooses what opponent to attack first. This is a strategic decision that may follow several strategies, e.g. to try to split the enemies in more, but weaker groups, or try to attack the enemy from the sides. In order to make the right decision, an analysis of the spatial (and health-related) positions of the opponent agents is needed.

3.4 The innovation of the system

The use of *separate potential fields* for the control of tactical, navigational, and strategic matters in a system of multiple units (our agents) in an RTS game has, as far as we know, not been described in academia before. Traditionally, A* and different types of state machines has been state-of-the-art in the gaming industry. Lately we have seen a growing interest for alternative solutions, partly as a result of the customer demand for more believable computer opponents, partly as a result of the increase in processing power that third generation game consoles such as Sony PlayStation 3 bring us. We believe that the use of both MAS techniques and potential fields (and why not our proposed combination of the two?) will gain ground as the game AI field matures. Lately, the performance of our solution has increased significantly compared to the results presented in Paper I and these late breaking improvements will of course be demonstrated.

3.5 The interactive aspects

Unfortunately, the human player interface is not yet released by the ORTS developers. If it will be available at the time of the conference, we will also be able to offer the audience to play games against our MAPF based bot. If not, we will illustrate its features through games against other computer opponents. We will be glad to illustrate the performance of our recently updated solution against the winner of the ORTS tournament described in Paper I.

There will be two windows updated in real time. The main window shows a 3D (see Figure 3.1), or 2D (see Figure 3.2) view of the units and the terrain. The second window (see Figure 3.3) shows the potential fields of a certain unit, as well as the resulting coordination done by the military commander. The whole potential field is shown here, although in the real application, only the potentials of the positions in the map that are considered interesting are calculated.

3.6 Conclusions

We will show a demonstration of a highly competitive game AI bot for the ORTS environment. It is built using the methodology described in Paper I and use a combination of Multi-agent coordination techniques and potential fields to try to win its games.

CHAPTER

FOUR

PAPER III

The Rise of Potential Fields in Real Time Strategy Bots

Johan Hagelbäck & Stefan J. Johansson

Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE). 2008.

4.1 Introduction

A *Real-time Strategy (*RTS*)* game is a game in which the players use resource gathering, base building, technological development and unit control in order to defeat their opponents, typically in some kind of war setting. The RTS game is not turn-based in contrast to board games such as Risk and Diplomacy. Instead, all decisions by all players have to be made in real-time. Generally the player has a top-down perspective on the battlefield although some 3D RTS games allow different camera angles. The real-time aspect makes the RTS genre suitable for multiplayer games since it allows players to interact with the game independently of each other and does not let them wait for someone else to finish a turn.

Khatib (1986) introduced a new concept while he was looking for a real-time obstacle avoidance approach for manipulators and mobile robots. The technique which he called *Artificial Potential Fields* moves a manipulator in a field of forces. The position to be reached is an attractive pole for the end effector (e.g. a robot) and obstacles are repulsive surfaces for the manipulator parts. Later on Arkin (1987) updated the knowledge by creating another technique using superposition of spatial vector fields in order to generate behaviours in his so called motor schema concept.

Many studies concerning potential fields are related to spatial navigation and obstacle avoidance, see e.g. Borenstein and Koren (1991); Massari et al. (2004). The technique is really helpful for the avoidance of simple obstacles even though they are numerous. Combined with an autonomous navigation approach, the result is even better, being able to surpass highly complicated obstacles (Borenstein & Koren, 1989).

Lately some other interesting applications for potential fields have been presented. The use of potential fields in architectures of multi agent systems is giving quite good results defining the way of how the agents interact. Howard et al. (2002) developed a mobile sensor network deployment using potential fields, and potential fields have been used in robot soccer (Johansson & Saffiotti, 2002; Röfer et al., 2004). Thurau et al. (2004b) has developed a game bot which learns reactive behaviours (or potential fields) for actions in the First-Person Shooter (FPS) game Quake II through imitation.

First we describe the domain followed by a description of our basic MAPF player. That solution is refined stepwise in a number of ways and for each and one of them we present the improvement shown in the results of the experiments. We then discuss the solution and conclude and show some directions of future work. In Paper I we have reported on the details of our methodology, and made a comparison of the computational costs of the bots, thus we refer to that study for these results.

4.2 ORTS

Open Real Time Strategy (ORTS) (Buro, 2007a) is a real-time strategy game engine developed as a tool for researchers within artificial intelligence (AI) in general and game AI in particular. ORTS uses a client-server architecture with a game server and players connected as clients. Each timeframe clients receive a data structure from the server containing the current game state. Clients can then issue commands for their units. Commands such as move unit A to (x, y) or attack opponent unit X with unit A. All client commands are executed in random order by the server.

Users can define different type of games in scripts where units, structures and their interactions are described. All type of games from resource gathering to full real time strategy (RTS) games are supported. We focus here on one type of two-player game, *Tankbattle*, which was one of the 2007 ORTS competitions (Buro, 2007a). In Tankbattle each player has 50 tanks and five bases. The goal is to destroy the bases of the opponent. Tanks are heavy units with long fire range and devastating firepower but a long cool-down period, i.e. the time after an attack before the unit is ready to attack again. Bases can take a lot of damage before they are destroyed, but they have no defence mechanism of their own so it may be important to defend own bases with tanks. The map in a tankbattle game has randomly generated terrain with passable lowland and impassable cliffs.

The game contains a number of neutral units (sheep). These are small indestructible units moving randomly around the map making pathfinding and collision detection more complex.

4.2.1 The Tankbattle competition of 2007

For comparison, the results from our original bot against the four top teams were reconstructed through running the matches again (see Table 4.1). To get a more detailed comparison than the win/lose ratio used in the tournament we introduce a game score. This score does not take wins or losses into consideration, instead it counts units and bases left after a game. The score for a game is calculated as:

$$\begin{aligned} score =& 5 \cdot (ownBasesLeft - oppBasesLeft) + \\ & ownUnitsLeft - oppUnitsLeft \end{aligned} \tag{4.1}$$

Team	Win %	Wins/games	Avg units	Avg bases	Avg score
NUS	0%	(0/100)	0.01	0.00	-46.99
WarsawB	0%	(0/100)	1.05	0.01	-42.56
UBC	24%	(24/100)	4.66	0.92	-17.41
Uofa.06	32%	(32/100)	4.20	1.45	-16.34
Average	14%	(14/100)	2.48	0.60	-30.83

Table 4.1: *Replication of the results of our bot in the ORTS tournament 2007 using the latest version of the ORTS server.*

4.2.2 Opponent descriptions

We refer to Paper I section 2.5.1 for opponent descriptions.

4.3 MAPF in ORTS, V.1

We have implemented an ORTS client for playing Tankbattle based on Multi-agent Potential Fields (MAPF) following the proposed methodology in Paper I. It includes the following six steps:

1. Identifying the objects

2. Identifying the fields
3. Assigning the charges
4. Deciding on the granularities
5. Agentifying the core objects
6. Construct the MAS architecture

Below we will describe the creation of our MAPF solution.

4.3.1 Identifying objects

We identify the following objects in our applications: Cliffs, Sheep, and own (and opponent) tanks, and base stations.

4.3.2 Identifying fields

We identified four tasks in ORTS Tankbattle: Avoid colliding with moving objects, Hunt down the enemy's forces, Avoid colliding with cliffs, and Defend the bases. This leads us to three types of potential fields: *Field of Navigation*, *Strategic Field*, and *Tactical field*.

The field of navigation is generated by repelling static terrain and may be pre-calculated in the initialisation phase. We would like agents to avoid getting too close to objects where they may get stuck, but instead smoothly pass around them.

The strategic field is an attracting field. It makes agents go towards the opponents and place themselves at appropriate distances from where they can fight the enemies.

Our own units, own bases and sheep generate small repelling fields. The purpose is that we would like our agents to avoid colliding with each other or bases as well as avoiding the sheep.

4.3.3 Assigning charges

Each unit (own or enemy), base, sheep and cliff have a set of charges which generate a potential field around the object. All fields generated by objects are weighted and summed to form a total field which is used by agents when selecting actions. The initial set of charges were found using trial and error. However, the order of importance between the objects simplifies the process of finding good values and the method seems robust enough to allow the bot to work good anyhow. We have tried to use traditional AI methods such as genetic algorithms to tune the parameters of the bot, but without success. The results of these studies are still unpublished. We used the following charges in the V.1 bot:[1]

[1] $I = [a, b[$ denote the half-open interval where $a \in I$, but $b \notin I$

The opponent units

$$p(d) = \begin{cases} k_1 d, & \text{if } d \in [0, MSD - a[\\ c_1 - d, & \text{if } d \in [MSD - a, MSD] \\ c_2 - k_2 d, & \text{if } d \in]MSD, MDR] \end{cases} \tag{4.2}$$

Unit	k_1	k_2	c_1	c_2	MSD	a	MDR
Tank	2	0.22	24.1	15	7	2	68
Base	3	0.255	49.1	15	12	2	130

Table 4.2: *The parameters used for the generic $p(d)$-function of Equation 4.2.*

Own bases Own bases generate a repelling field for obstacle avoidance. Below in Equation 4.3 is the function for calculating the potential $p_{ownB}(d)$ at distance d (in tiles) from the center of the base.

$$p_{ownB}(d) = \begin{cases} 5.25 \cdot d - 37.5 & \text{if } d <= 4 \\ 3.5 \cdot d - 25 & \text{if } d \in]4, 7.14] \\ 0 & \text{if } d > 7.14 \end{cases} \tag{4.3}$$

The own tanks The potential $p_{ownU}(d)$ at distance d (in tiles) from the center of an own tank is calculated as:

$$p_{ownU}(d) = \begin{cases} -20 & \text{if } d <= 0.875 \\ 3.2d - 10.8 & \text{if } d \in]0.875, l], \\ 0 & \text{if } d >= l \end{cases} \tag{4.4}$$

Sheep Sheep generate a small repelling field for obstacle avoidance. The potential $p_{sheep}(d)$ at distance d (in tiles) from the center of a sheep is calculated as:

$$p_{sheep}(d) = \begin{cases} -10 & \text{if } d <= 1 \\ -1 & \text{if } d \in]1, 2] \\ 0 & \text{if } d > 2 \end{cases} \tag{4.5}$$

Figure 3.2 in Paper II shows a 2D view of the map during a tankbattle game. It shows our agents (green) moving in to attack enemy bases and units (red). Figure 3.3 shows the potential field view of the same tankbattle game. Dark areas has low potential and light areas high potential. The light ring around enemy bases and units, located at maximum shooting distance of our tanks, is the distance our agents prefer to attack opponent units from. It is the final move goal for our units.

4.3.4 Granularity

We believed that tiles of 8*8 positions was a good balance between performance on the one hand, and the time it would take to make the calculations, on the other.

4.3.5 Agentifying and the construction of the MAS

We put one agent in each unit, and added a coordinator that took care of the coordination of fire. For details on the implementation description we have followed, we refer to Paper I.

4.4 Weaknesses and counter-strategies

To improve the performance of our bot we observed how it behaved against the top teams from the 2007 years' ORTS tournament. From the observations we have defined a number of weaknesses of our bot and proposed solutions to these. For each improvement we have run 100 games against each of the teams NUS, WarsawB, UBC and Uofa.06. A short description of the opponent bots can be found below. The experiments are started with a randomly generated seed and then two games, one where our bot is team 0 and one where our bot is team 1, are played. For the next two games the seed is incremented by 1, and the experiments continues in this fashion until 100 games are played.

By studying the matches, we identified four problems with our solution:

1. Some of our units got stuck in the terrain due to problems finding their way through narrow passages.

2. Our units exposed themselves to hostile fire during the cool down phase.

3. Some of the units were not able to get out of local minima created by the potential field.

4. Our units came too close to the nearest opponents if the opponent units were gathered in large groups.

We will now describe four different ways to address the identified problems by adjusting the original bot V.1 described in Paper I. The modifications are listed in Table 4.3.

Properties	V.1	V.2	V.3	V.4	V.5
Full resolution		$\surd$	$\surd$	$\surd$	$\surd$
Defensive field			$\surd$	$\surd$	$\surd$
Charged pheromones				$\surd$	$\surd$
Max. potential strategy					$\surd$

Table 4.3: *The implemented properties in the different experiments using version 1–5 of the bot.*

4.4.1 Increasing the granularity, V.2

In the original ORTS bot we used 128x128 tiles for the potential field, where each tile was 8x8 positions in the game world. The potential field generated from a game object, for example own tanks, was pre-calculated in 2-dimensional arrays and simple copied at runtime into the total potential field. This resolution proved not to be detailed enough. In the tournament our units often got stuck in terrain or other obstacles such as our own bases. This became a problem, since isolated units are easy targets for groups of attacking units.

The proposed solution is to increase the resolution to 1x1 positions per tile. To reduce the memory requirements we do not pre-calculate the game object potential fields, instead the potentials are calculated at runtime by passing the distance between an own unit and each object to a mathematical formula. To reduce computation time we only calculate the potentials in the positions around each own unit, not the whole total potential field as in the original bot. Note that the static terrain is still pre-calculated and constructed using 8x8 positions tiles. Below is a description and formulas for each of the fields. In the experiments we use weight $1/7 \approx 0.1429$ for each of the weights w_1 to w_7. The weight w_7 is used to weight the terrain field which, except for the weight, is identical to the terrain field used in the original bot. The results from the experiments are presented in Table 4.4. Below is a detailed description of the fields.

Team	Win %	Wins/games	Avg units	Avg bases	Avg score
NUS	9%	(9/100)	1.18	0.57	-32.89
WarsawB	0%	(0/100)	3.03	0.12	-36.71
UBC	24%	(24/100)	16.11	0.94	0.46
Uofa.06	42%	(42/100)	10.86	2.74	0.30
Average	18.75%	(18.75/100)	7.80	1.09	-17.21

Table 4.4: *Experiment results from increasing the granularity.*

The opponent units and bases. All opponent units and bases generate symmetric surrounding fields where the highest potentials surround the objects at radius D, the

MSD, R refers to the *Maximum Detection Range*, the distance from which an agent starts to detect the opponent unit. The potentials $p_{oppU}(d)$ and $p_{oppB}(d)$ at distance d from the center of an agent are calculated as:

$$p_{oppU}(d) = w_1 \cdot \begin{cases} 240/d(D-2), & \text{if } d \in [0, D-2[\\ 240, & \text{if } d \in [D-2, D] \\ 240 - 0.24(d-D) & \text{if } d \in]D, R] \end{cases} \tag{4.6}$$

$$p_{oppB}(d) = w_6 \cdot \begin{cases} 360/(D-2) \cdot d, & \text{if } d \in [0, D-2[\\ 360, & \text{if } d \in [D-2, D] \\ 360 - (d-D) \cdot 0.32 & \text{if } d \in]D, R] \end{cases} \tag{4.7}$$

Own units — tanks. Own units generate repelling fields for obstacle avoidance. The potential $p_{ownU}(d)$ at distance d from the center of a unit is calculated as:

$$p_{ownU}(d) = w_3 \cdot \begin{cases} -20 & \text{if } d <= 14 \\ 32 - 2 \cdot d & \text{if } d \in]14, 16] \end{cases} \tag{4.8}$$

Own bases. Own bases also generate repelling fields for obstacle avoidance. Below is the function for calculating the potential $p_{ownB}(d)$ at distance d from the center of the base.

$$p_{ownB}(d) = w_4 \cdot \begin{cases} 6 \cdot d - 258 & \text{if } d <= 43 \\ 0 & \text{if } d > 43 \end{cases} \tag{4.9}$$

Sheep. Sheep generate a small repelling field for obstacle avoidance. The potential $p_{sheep}(d)$ at distance d from the center of a sheep is calculated as:

$$p_{sheep}(d) = w_5 \cdot \begin{cases} -20 & \text{if } d <= 8 \\ 2 \cdot d - 25 & \text{if } d \in]8, 12.5] \end{cases} \tag{4.10}$$

4.4.2 Adding a defensive potential field, V.3

After a unit has fired its weapon the unit has a cooldown period when it cannot attack. In the original bot our agents was, as long as there were enemies within MSD (D), stationary until they were ready to fire again. The cooldown period can instead be used for something more useful and we propose the use of a defensive field. This field makes the units retreat when they cannot attack, and advance when they are ready to attack once again. With this enhancement our agents always aim to be at D of the closest opponent unit or base and surround the opponent unit cluster at D. The potential $p_{def}(d)$ at distance d from the center of an agent is calculated using the formula in Equation 4.11. The results from the experiments are presented in Table 4.5.

$$p_{def}(d) = w_2 \cdot \begin{cases} w_2 \cdot (-800 + 6.4 \cdot d) & \text{if } d <= 125 \\ 0 & \text{if } d > 125 \end{cases} \tag{4.11}$$

Team	Win %	Wins/games	Avg units	Avg bases	Avg score
NUS	64%	(64/100)	22.95	3.13	28.28
WarsawB	48%	(48/100)	18.32	1.98	15.31
UBC	57%	(57/100)	30.48	1.71	29.90
Uofa.06	88%	(88/100)	29.69	4.00	40.49
Average	64.25%	(64.25/100)	25.36	2.71	28.50

Table 4.5: *Experiment results from adding a defensive field.*

4.4.3 Adding charged pheromones, V.4

The local optima problem is well known in general when using PF. Local optima are positions in the potential field that has higher potential than all its neighbouring positions. A unit positioned at a local optimum will therefore get stuck even if the position is not the final destination for the unit. In the original bot agents that had been idle for some time moved in a random direction for some frames. This is not a very reliable solution to the local optima problem since there is not guarantee that the agent has moved out of, or will not directly return to, the local optima.

Thurau, Bauckhage, and Sagerer (2004a) described a solution to the local optima problem called avoid-past potential field forces. In this solution each agent generates a trail of negative potentials on previous visited positions, similar to a pheromone trail used by ants. The trail pushes the agent forward if it reaches a local optima.

We have introduced a trail that adds a negative potential to the last 20 positions of each agent. Note that an agent is not effected by the trails of other own agents. The negative potential for the trail was set to -0.5 and the results from the experiments are presented in Table 4.6.

Team	Win %	Wins/games	Avg units	Avg bases	Avg score
NUS	73%	(73/100)	23.12	3.26	32.06
WarsawB	71%	(71/100)	23.81	2.11	27.91
UBC	69%	(69/100)	30.71	1.72	31.59
Uofa.06	93%	(93/100)	30.81	4.13	46.97
Average	76.5%	(76.5/100)	27.11	2.81	34.63

Table 4.6: *Experiment results from adding charged pheromones.*

4.4.4 Using maximum potentials, V.5

In the original bot all potential fields generated from opponent units were weighted and summed to form the total potential field which is used for navigation by our agents. The effect of summing the potential fields generated by opponent units is that the highest potentials are generated from the centre of the opponent unit cluster. This makes our agents attack the centre of the enemy force instead of keeping the MSD to the closest enemy. The proposed solution to this issue is that, instead of summing the potentials generated by opponent units and bases, we add the highest potential any opponent unit or base generates. The effect of this is that our agents engage the closest enemy unit at maximum shooting distance instead of moving towards the centre of the opponent unit cluster. The results from the experiments are presented in Table 4.7.

Team	Win %	Wins/games	Avg units	Avg bases	Avg score
NUS	100%	(100/100)	28.05	3.62	46.14
WarsawB	99%	(99/100)	31.82	3.21	47.59
UBC	98%	(98/100)	33.19	2.84	46.46
Uofa.06	100%	(100/100)	33.19	4.22	54.26
Average	99.25%	(99.25/100)	31.56	3.47	48.61

Table 4.7: *Experiment results from using maximum potential, instead of summing the potentials.*

4.5 Discussion

The results clearly show that the improvements we suggest increases the performance of our solution dramatically. We will now discuss these improvements from a wider perspective, asking ourselves if it would be easy to achieve the same results without using potential fields.

4.5.1 Using full resolution

We believed that the PF based solution would suffer from being slow. Because of that, we did not initially use the full resolution of the map. However, we do so now, and by only calculating the potentials in a number of move candidates for each unit (rather than all positions of the map), we have no problems at all to let the units move in full resolution. This also solved our problems with units getting stuck at various objects and having problems to go through narrow passages.

4.5.2 Avoiding the obstacles

The problems with local optima are well documented for potential fields. It is a result of the lack of planning. Instead, a one step look-ahead is used in a reactive manner. This is of course problematic in the sense that the unit is not equipped to plan its way out of a sub-optimal position. It will have to rely on other mechanisms. The pheromone trail is one such solution that we successfully applied to avoid the problem.

On the other hand, there are also advantages of avoiding to plan, especially in a dynamically changing environment where long term planning is hard.

4.5.3 Avoiding opponent fire

The trick to avoid opponent fire by adding a defensive potential field during the cool-down phase is not hard to implement in a traditional solution. By adding a state of cool-down, which implements a flee behaviour, that makes the unit run away from the enemies, that could be achieved. The potential problem here is that it may be hard to coordinate such a movement with other units trying to get to the front, so some sort of coordinating mechanism may be needed. While this mechanism is implicit in the PF case (through the use of small repulsive forces between the own units), it will have to be taken care of explicitly in the planning case.

4.5.4 Staying at maximum shooting distance

The problem we had, to keep the units at the MSD from the *nearest* opponent, was easily solved by letting that opponent be the one setting the potential in the opponent field, rather than the gravity of the whole opponent group (as in the case of summing all potentials). As for the case of bots using planning, we can not see that this really is a problem for them.

4.5.5 On the methodology

We have used the newer version of the ORTS server for the experiments. On the one hand, it allows us to use the latest version of our bot, which of course is implemented to work with the new server. On the other hand, we could not get one of the last years' participants to work with the new server. Since games like these are not transitive in the sense that if player A wins over player B, and player B wins over player C, then player A will not be guaranteed to win over player C, there is a risk that the bot that was left out of these experiments would have been better than our solution. However, the point is that we have shown that a potential field-based player is able to play significantly better than a number of planning-based counterparts. Although we have no reason to believe that the UofA07 bot would be an exception, we do not have the results to back it up.

The order of the different versions used was determined after running a small series of matches with different combinations of improvements added. We then picked them in the order that best illustrated the effects of the improvements.

However, our results were further validated in the 2008 ORTS tournament, where our PF based bots won the three competitions that we participated in (Collaborative Pathfinding, Tankbattle, and Complete RTS). In the Tankbattle competition, we won all 100 games against NUS, the winner of last year, and only lost four of 100 games to Lidia (see Table 4.8).

Team	Total win %	Blekinge	Lidia	NUS
Blekinge	98	—	96	100
Lidia	43	4	—	82
NUS	9	0	18	—

Table 4.8: *Results from the ORTS Tankbattle 2008 competition.*

4.6 Conclusions and Future Work

We have presented a five step improvement of a potential field based bot that plays the Strategic Combat game in ORTS. By the full improvement we managed to raise the performance from winning less than 7 per cent to winning more than 99 per cent of the games against four of the top five teams at the ORTS tournament 2007. Our bot did also quite easily win the 2008 tournament.

We believe that potential fields is a successful option to the more conventional planning-based solutions that uses e.g. A* in Real Time Strategy games.

In the future, we will report on the application of the methodology described in Paper I to a number of other ORTS games. We will also set up a new series of experiments where we adjust the ability/efficiency trade-off of the bot in real time to increase the player experience.

CHAPTER

FIVE

PAPER IV

Dealing with Fog of War in a Real Time Strategy Game Environment

Johan Hagelbäck & Stefan J. Johansson

Proceedings of 2008 IEEE Symposium on Computational Intelligence and Games (CIG). 2008.

5.1 Introduction

A *Real-time Strategy (*RTS*)* game is a game in which the players use resource gathering, base building, technological development and unit control in order to defeat their opponents, typically in some kind of war setting. An RTS game is not turn-based in contrast to board games such as Risk and Diplomacy. Instead, all decisions by all players have to be made in real-time. The player usually has an isometric birds view perspective of the battlefield although some 3D RTS games allow different camera angles. The real-time aspect makes the RTS genre suitable for multiplayer games since it allows players to interact with the game independently of each other and does not let them wait for someone else to finish a turn.

In RTS games computer bots often *cheat* in the sense that they get access to complete visibility (perfect information) of the whole game world, including the positions of the opponent units. Cheating is, according to Nareyek (2004), *"very annoying for the player* *if discovered"* and he predicts the game AIs to get a larger share of the processing power in the future which in turn may open up for the possibility to use more sophisticated AIs.

We will show how a bot that uses potential fields can be modified to deal with imperfect information, i.e. the parts of the game world where no own units are present are unknown (usually referred to as Fog of War, or FoW). We will also show that our modified bot with imperfect information, named FoWbot, actually not only perform equally good, compared to a version with perfect information (called PIbot), but also that it at an average consumes *less* computational power than its cheating counterpart.

5.1.1 Research Question and Methodology

The main research question of this paper is: *Is it possible to construct a bot without access to perfect information for* RTS *games that perform as well as bots that have perfect information?* This breaks down to:

1. What is the difference in performance between using a FoWbot compared to a PIbot in terms of a) the number of won matches, and b) the number of units and bases left if the bot wins?
2. To what degree will a field of exploration help the FoW bot to explore the unknown environment?
3. What is the difference in the computational needs for the FoWbot compared to the PIbot?

In order to approach the research questions above, we will implement a FoW version of our original PIbot and compare its performance, exploration and processing needs with the original.

5.1.2 Outline

First we describe the domain followed by a description of our Multi-agent Potential Field (MAPF) player. In the next section we describe the adjustments needed to implement a working FoW bot and then we present the experiments and their results. We finish by discussing the results, draw some conclusions and line out possible directions for future work.

5.2 ORTS

Open Real Time Strategy (ORTS) (Buro, 2007a) is a real-time strategy game engine developed as a tool for researchers within AI in general and game AI in particular. ORTS uses a client-server architecture with a game server and players connected as clients.

Each timeframe clients receive a data structure from the server containing the current state of the game. Clients can then activate their units in various ways by sending commands to them. These commands can be like *move unit A to* (x, y) or *attack opponent unit X with unit A*. All client commands for each time frame are sent in parallel, and executed in random order by the server.

Users can define different types of games in scripts where units, structures and their interactions are described. All types of games from resource gathering to full real time strategy (RTS) games are supported. We focus here on one type of two-player game, *Tankbattle*, which was one of the 2007 ORTS competitions (Buro, 2007a).

In *Tankbattle* each player has 50 tanks and five bases. The goal is to destroy the bases of the opponent. Tanks are heavy units with long fire range and devastating firepower but a long cool-down period, i.e. the time after an attack before the unit is ready to attack again. Bases can take a lot of damage before they are destroyed, but they have no defence mechanism so it may be important to defend own bases with tanks. The map in a Tankbattle game has randomly generated terrain with passable lowland and impassable cliffs.

The game contains a number of neutral units (sheep). These are small, indestructible units moving randomly around the map. The purpose of them is to make pathfinding and collision detection more complex.

We have in our experiments chosen to use an environment based on the best participants of the last year's ORTS tournament (Buro, 2007b).

5.2.1 Descriptions of Opponents

We refer to Paper I section 2.5.1 for opponent descriptions.

5.3 Multi-agent Potential Fields

Khatib (1986) introduced a new concept while he was looking for a real-time obstacle avoidance approach for manipulators and mobile robots. The technique, which he called *Artificial Potential Fields*, moves a manipulator in a field of forces. The position to be reached is an attractive pole for the end effector (e.g. a robot) and obstacles are repulsive surfaces for the manipulator parts. Later on Arkin (1987) updated the knowledge by creating another technique using superposition of spatial vector fields in order to generate behaviours in his so called motor schema concept.

Many studies concerning potential fields are related to spatial navigation and obstacle avoidance, see e.g. Borenstein and Koren (1991); Massari et al. (2004). The technique is really helpful for the avoidance of simple obstacles even though they are numerous. Combined with an autonomous navigation approach, the result is even better, being able to surpass highly complicated obstacles (Borenstein & Koren, 1989).

Lately some other interesting applications for potential fields have been presented. The use of potential fields in architectures of multi agent systems has shown promising results. Howard et al. (2002) developed a mobile sensor network deployment using potential fields, and potential fields have been used in robot soccer (Johansson & Saffiotti, 2002; Röfer et al., 2004). Thurau et al. (2004b) has developed a game bot which learns reactive behaviours (or potential fields) for actions in the First-Person Shooter (FPS) game Quake II through imitation.

In Paper I we propose a methodology for creating an RTS game bot based on Multi-agent Potential Fields (MAPF). This bot was further improved in Paper III and it is the improved version that we have used in this experiment.

5.4 MAPF in ORTS

We have implemented an ORTS client for playing Tankbattle games based on Multi-agent Potential Fields (MAPF) following the proposed methodology of Paper I. It includes the following six steps:

1. Identifying the objects
2. Identifying the fields
3. Assigning the charges
4. Deciding on the granularities
5. Agentifying the core objects
6. Construct the MAS Architecture

Below we will describe the creation of our MAPF solution.

5.4.1 Identifying objects

We identify the following objects in our applications: Cliffs, Sheep, and own (and opponent) tanks, and base stations.

5.4.2 Identifying fields

We identified five tasks in ORTS Tankbattle:

- Avoid colliding with moving objects,
- avoid colliding with cliffs, and
- find the enemy,

- destroy the enemy's forces, and
- defend the bases.

The latter task will not be addressed in this study (instead, see Paper III), but the rest lead us to three types of potential fields: *Field of navigation*, *Strategic field*, *Tactical field*, and *Field of exploration*.

The field of navigation is generated by repelling static terrain and may be pre-calculated in the initialisation phase. We would like agents to avoid getting too close to objects where they may get stuck, but instead smoothly pass around them.

The strategic field is an attracting field. It makes agents go towards the opponents and place themselves at appropriate distances from where they can fight the enemies.

Our own units, own bases and the sheep generate small repelling fields. The purpose is that we would like our agents to avoid colliding with each other or the bases as well as avoiding the sheep.

The field of exploration helps the units to explore unknown parts of the game map. Since it is only relevant in the case we have incomplete information, it is not part of the PIbot that we are about to describe now. More information about the field of exploration is found in Section 5.5.3.

5.4.3 Assigning charges

Each unit (own or enemy), base, sheep and cliff have a set of charges which generate a potential field around the object. All fields generated by objects are weighted and summed to form a total field which is used by agents when selecting actions. The initial set of charges were found using trial and error. However, the order of importance between the objects simplifies the process of finding good values and the method seems robust enough to allow the bot to work good anyhow. We have tried to use traditional AI methods such as genetic algorithms to tune the parameters of the bot, but without success. We used the following charges in the PIbot:[1]

The opponent units

$$p(d) = \begin{cases} k_1 d, & \text{if } d \in [0, MSD - a[\\ c_1 - d, & \text{if } d \in [MSD - a, MSD] \\ c_2 - k_2 d, & \text{if } d \in]MSD, MDR] \end{cases} \tag{5.1}$$

[1] $I = [a, b[$ denote the half-open interval where $a \in I$, but $b \notin I$

Unit	k_1	k_2	c_1	c_2	MSD	a	MDR
Tank	2	0.22	24.1	15	7	2	68
Base	3	0.255	49.1	15	12	2	130

Table 5.1: *The parameters used for the generic $p(d)$-function of Equation 5.1.*

Own bases Own bases generate a repelling field for obstacle avoidance. Below in Equation 5.2 is the function for calculating the potential $p_{ownB}(d)$ at distance d (in tiles) from the center of the base.

$$p_{ownB}(d) = \begin{cases} 5.25 \cdot d - 37.5 & \text{if } d <= 4 \\ 3.5 \cdot d - 25 & \text{if } d \in]4, 7.14] \\ 0 & \text{if } d > 7.14 \end{cases} \tag{5.2}$$

The own tanks The potential $p_{ownU}(d)$ at distance d (in tiles) from the center of an own tank is calculated as:

$$p_{ownU}(d) = \begin{cases} -20 & \text{if } d <= 0.875 \\ 3.2d - 10.8 & \text{if } d \in]0.875, l], \\ 0 & \text{if } d >= l \end{cases} \tag{5.3}$$

Sheep Sheep generate a small repelling field for obstacle avoidance. The potential $p_{sheep}(d)$ at distance d (in tiles) from the center of a sheep is calculated as:

$$p_{sheep}(d) = \begin{cases} -10 & \text{if } d <= 1 \\ -1 & \text{if } d \in]1, 2] \\ 0 & \text{if } d > 2 \end{cases} \tag{5.4}$$

Figure 3.2 in Paper II shows a 2D view of the map during a tankbattle game. It shows our agents (green) moving in to attack enemy bases and units (red). Figure 3.3 shows the potential field view of the same tankbattle game. Dark areas has low potential and light areas high potential. The light ring around enemy bases and units, located at maximum shooting distance of our tanks, is the distance our agents prefer to attack opponent units from. It is the final move goal for our units.

5.4.4 Finding the right granularity

Concerning the granularity, we use full resolution (down to the point level) but only evaluate eight directions in addition to the position where the unit is. However, this is done in each time frame for each of our units.

5.4.5 Agentifying the objects

We put one agent in every own unit able to act in some way (thus, the bases are excluded). We have chosen not to simulate the opponent using agents, although that may be possible, it is outside the scope of this experiment.

5.4.6 Constructing the MAS

All of our unit agents are communicating with a common *interface agent* to get and leave information about the state of the game such as to get the position of (visible) opponents, and to submit the actions taken by our units. The bot also has an *attack coordinating agent* that points out what opponent units to attack, if there are several options.

Attack coordination

We use a coordinator agent to globally optimise attacks at opponent units. The coordinator aims to destroy as many opponent units as possible each frame by concentrating fire on already damaged units. The attack coordinator used are identical to the attack coordinator agent described in Paper I section 2.4.6.

5.5 Modifying for the Fog of War

To enable FoW for only one client, we made a minor change in the ORTS server. We added an extra condition to an IF statement that always enabled fog of war for client 0. Due to this, our client is always client 0 in the experiments (of course, it does not matter from the game point of view if the bots play as client 0 or client 1).

To deal with fog of war we have made some changes to the bot described in Paper III. These changes deal with issues like remember locations of enemy bases, explore unknown terrain to find enemy bases and units, and to remember the terrain (i.e. the positions of the impassable cliffs at the map) even when there are no units near. Another issue is dealing with performance since these changes are supposed to require more runtime calculations than the PIbot. Below are proposed solutions to these issues.

5.5.1 Remember Locations of the Enemies

In ORTS a data structure with the current game world state is sent each frame from the server to the connected clients. If fog of war is enabled, the location of an enemy base is only included in the data structure if an own unit is within visibility range of the base. It means that an enemy base that has been spotted by an own unit and that unit is destroyed, the location of the base is no longer sent in the data structure. Therefore our bot has a dedicated global map agent to which all detected objects are reported. This agent always remembers the location of previously spotted enemy bases until a base is destroyed, as well as distributes the positions of detected enemy tanks to all the own units.

The global map agent also takes care of the map sharing concerning the opponent tank units. However, it only shares momentary information about opponent tanks that are within the detection range of at least one own unit. If all units that see a certain opponent tank are destroyed, the position of that tank is no longer distributed by the global map agent and that opponent disappears from our map.

5.5.2 Dynamic Knowledge about the Terrain

If the game world is completely known, the knowledge about the terrain is static throughout the game. In the original bot, we created a static potential field for the terrain at the beginning of each new game. With fog of war, the terrain is partly unknown and must be explored. Therefore our bot must be able to update its knowledge about the terrain.

Once the distance to the closest impassable terrain has been found, the potential is calculated as:

$$p_{terrain}(d) = \begin{cases} -10000 & \text{if } d <= 1 \\ -5/(d/8)^2 & \text{if } d \in]1, 50] \\ 0 & \text{if } d > 50 \end{cases} \tag{5.5}$$

5.5.3 Exploration

Since the game world is partially unknown, our units have to explore the unknown terrain to locate the hidden enemy bases. The solution we propose is to assign an attractive field to each unexplored game tile. This works well in theory as well as in practice if we are being careful about the computation resources spent on it.

The potential $p_{unknown}$ generates in a point (x, y) is calculated as follows:

1. Divide the terrain tile map into blocks of 4x4 terrain tiles.

2. For each block, check every terrain tile in the block. If the terrain is unknown in ten or more of the checked tiles, the whole block is considered unknown.

3. For each block that needs to be explored, calculate the Manhattan Distance md from the center of the own unit to the center of the block.

4. Calculate the potential $p_{unknown}$ each block generates using Equation 5.6 below.

5. The total potential in (x, y) is the sum of the potentials each block generates in (x, y).

$$p_{unknown}(md) = \begin{cases} (0.25 - \frac{md}{8000}) & \text{if } md <= 2000 \\ 0 & \text{if } md > 2000 \end{cases} \quad (5.6)$$

5.6 Experiments

We have conducted three sets of experiments:

1. Show the performance of FoWbot playing against bots with perfect information.
2. Show the impact of the field of exploration in terms of the detected percentage of the map.
3. Show computational resources needed for FoWbot compared to the PIbot.

5.6.1 Performance

To show the performance of our bot we have run 100 games against each of the top teams NUS, WarsawB, UBC and Uofa.06 from the 2007 years ORTS tournament as well as 100 matches against our PIbot. In the experiments the first game starts with a randomly generated seed, and the seed is increased by 1 for each game played. The same start seed is used for all four opponents.

The experiment results presented in Table 5.2 shows that our MAPF based FoWbot wins over 98% of the games even though our bot has imperfect information and the opponent bots have perfect information about the game world.

We may also see that when PIbot and FoWbot are facing each other, FoWbot wins (surprisingly enough) about twice as often as PIbot. We will come back to the analysis of these results in the discussion.

	FoWbot			PIbot		
Team	Win %	Units	Base	Win %	Units	Base
NUS	100%	29.74	3.62	100%	28.05	3.62
WarsawB	98%	32.35	3.19	99%	31.82	3.21
UBC	96%	33.82	3.03	98%	33.19	2.84
Uofa.06	100%	34.81	4.27	100%	33.19	4.22
Average	98.5%	32.68	3.53	99.25%	31.56	3.47
FoWbot	—	—	—	66%	9.37	3.23
PIbot	34%	4.07	1.81	—	—	—

Table 5.2: *Performance of FoWbot and PIbot in 100 games against five opponents.*

5.6.2 The Field of Exploration

We ran 20 different games in this experiment, each where the opponent faced both a FoWbot with the field of exploration enabled, and one where this field was disabled (the rest of the parameters, seeds, etc. were kept identical).

Figure 5.1 shows the performance of the exploration field. It shows how much area, for both types of bots, that is explored, given how long a game has proceeded. The standard deviation increases with the time since only a few of the games last longer than three minutes.

In Table 5.3, we see that the use of the field of exploration (as implemented here) does not improve the results dramatically. The differences are not statistically significant.

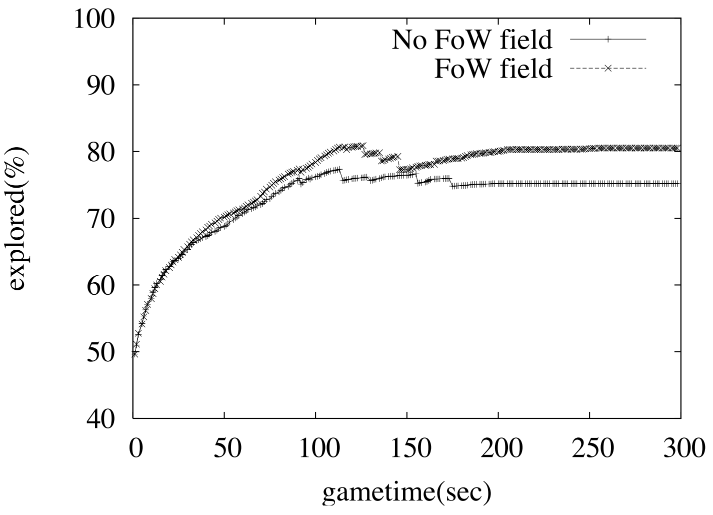


Figure 5.1: *The average explored area given the current game time for a bot using the field of exploration, compared to one that does not.*

Version	Won	Lost	Avg. Units	Avg. Bases
With FoE	20	0	28.65	3.7
Without FoE	19	1	27.40	3.8

Table 5.3: *Performance of the FoWbot with and without Field of Exploration (FoE) in 20 matches against NUS.*

5.6.3 Computational Resources

To show the computational resources needed we have run 100 games using the PIbot against team NUS and 100 games with the same opponent using the FoWbot. The same seeds are used in both series of runs. For each game we measured the average time (in milliseconds) that the bot uses in each game frame and the number of own units left. Figure 5.2 shows the average time for both our bots in relation to number of own units left.

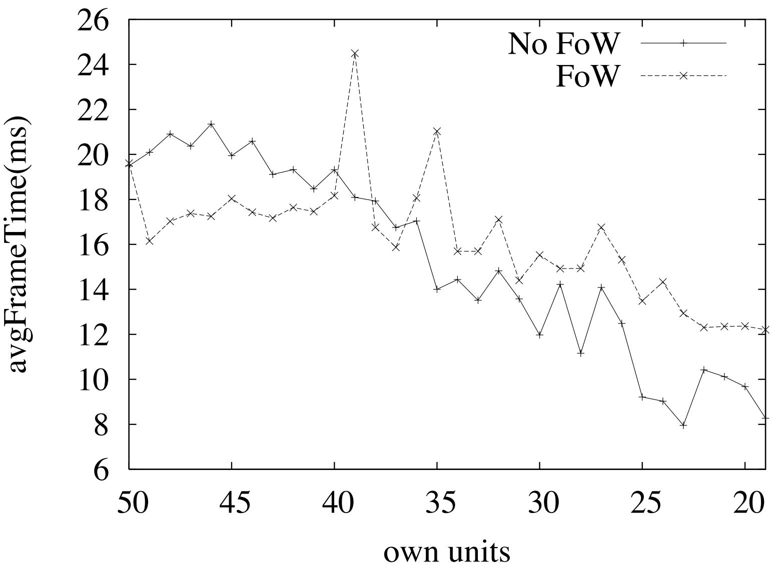


Figure 5.2: *The average frame time used for PIbot and FoWbot against team NUS.*

5.7 Discussion

The performance shows good results, but the question remains: could it be better without FoW? We ran identical experiments which showed that the average winning percentage was slightly higher for the PIbot compared to the FoWbot when they faced the top teams of ORTS 2007, see Table 5.2. We can also see that the number of units, as well as bases left are marginally higher for the FoWbot compared to the PIbot. However these results are not statistically significant.

Where we actually see a clear difference is when PIbot meets FoWbot and surprisingly enough FoWbot wins 66 out of 100 games. We therefore have run a second series of 100 matches with a version of the PIbot where maximum detection range (i.e. the range at which a bot starts to sense the opponents' potential field) was decreased from 1050 to 450. This is not the same as the *visibility range* in the FoWbot (which is just 160). Remember that the FoWbot has a global map agent that helps the units to distribute the positions of visible enemies to units that do not have visual contact with the enemy unit in question. However, the decrease of the maximum detection range in PIbot makes it less prone to perform single unit attacks and the FoWbot only wins 55 out of 100 games in our new series of matches, which leaves a 37% probability that PIbot is the better of the two (compared to 0.2% in the previous case).

In Figure 5.1 we see that using the field of exploration in general gives a higher degree of explored area in the game, but the fact that the average area is not monotonically increasing as the games go on may seem harder to explain. One plausible explanation is that the games where our units do not get stuck in the terrain will be won faster as well as having more units available to explore the surroundings. When these games end, they do not contribute to the average and the average difference in explored areas will decrease.

Does the field of exploration contribute to the performance? Is it at all important to be able to explore the map? Our results (see Table 5.3) indicate that it in this case may not be that important. However, the question is complex. Our experiments were carried out with an opponent bot that had perfect information and thus was able to find our units. The results may have been different if also the opponent lacked perfect information.

Concerning the processor resources, the average computational effort is initially higher for the PIbot. The reason for that is that it knows the positions of all the opponent units, thus include all of them in the calculations of the strategic potential field. As the number of remaining units decrease the FoWbot has a slower decrease in the need for computational power than the PIbot. This is because there is a comparably high cost to keep track of the terrain and the field of navigation that it generates, compared to having it static as in the case of the PIbot.

This raise the question of whether having access to perfect information is an advantage compared to using a FoWbot. It seems to us, at least in this study, that it is not at all the case. Given that we have at an average around 32 units left when the game ends, the average time frame probably requires more from the PIbot, than from the FoWbot. However, that will have to be studied further before any general conclusions may be drawn in that direction.

Finally some comments on the methodology of this study. There are of course details that could have been adjusted in the experiments in order to e.g. balance the performance of PIbot vs FoWbot. As an example, by setting the detection range in the PIbot identical to the one in the FoWbot and at the same time add the global map agent (that is only used in the FoWbot today) to the PIbot. However, it would significantly increase the computational needs of the PIbot to do so. We are of course eager to improve our bots as far as possible (for the next ORTS competition 2009; a variant of our PIbot won the 2008 competition in August with a win percentage of 98%), and every detail that may improve it should be investigated.

5.8 Conclusions and Future Work

Our experiments show that a MAPF based bot can be modified to handle imperfect information about the game world, i.e. FoW. Even when facing opponents with perfect information our bot wins over 98% of the games. The FoWbot requires about the same computational resources as the PIbot, although it adds a field of exploration that increases the explored area of the game.

Future work include a more detailed experiment regarding the computational needs as well as an attempt to utilise our experiences from these experiments in the next ORTS tournament, especially the feature that made FoWbot beat PIbot.

CHAPTER
SIX

PAPER V

A Multi-agent Potential Field based bot for a Full RTS Game Scenario.

Johan Hagelbäck & Stefan J. Johansson
Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2009.

6.1 Introduction

There are many challenges for a real-time strategy (RTS) bot. The bot has to control a number of units performing tasks such as gathering resources, exploring the game world, hunting down the enemy and defend own bases. In modern RTS games, the number of units can in some cases be up to several hundred. The highly dynamic properties of the game world (e.g. due to the large number of moving objects) make navigation sometimes difficult using conventional pathfinding methods.

Artificial Potential Fields, an area originating from robotics, has been used with some success in video games. Thurau et al. has developed a game bot which learns behaviours in the First-Person Shooter game Quake II through imitation (Thurau et al., 2004b). The behavious are represented as attractive potential fields placed at interesting points in the game world, for example choke points or areas providing cover. The strength of the fields are increased/decreased by observing a human player.

6.1.1 Multi-agent Potential Fields

In previous work (see Paper I) we proposed a methodology for designing a multi-agent potential fields (MAPF) based bot in a real-time strategy game environment. The methodology involved the following six steps: *i) Identifying the objects, ii) Identifying the fields, iii) Assigning the charges, iv) Deciding on the granularities, v) Agentifying the core objects* and *vi) Construct the* MAS *architecture.* For further details on the methodology, we refer to the original description in Paper I. In this paper we use the methodology to build a bot for the full RTS game scenario.

6.2 ORTS

Open Real Time Strategy (ORTS) (Buro, 2007a) is a real-time strategy game engine developed as a tool for researchers within AI in general and game AI in particular. ORTS uses a client-server architecture with a game server and players connected as clients. Users can define different types of games in scripts where units, structures and their interactions are described. All types of games from resource gathering to full real time strategy (RTS) games are supported.

In previous work (see Paper I and Paper III) we used the proposed methodology to develop a MAPF based bot for the quite simple game type *Tankbattle*. Here, we extend the work to handle the more complex Full RTS game (Buro, 2007a). In this game, two players start with five workers and a control center each. The *workers* can be used to gather resources from nearby mineral patches, or to construct new control centers, barracks or factories. A *control center* serves as the drop point for resources gathered by workers, and it can produce new workers as well. *Barracks* are used to construct *marines;* light-weight combat units. If a player has at least one barrack, it can construct a *factory*. Factories are used to construct *tanks;* heavy combat units with long firerange. A player wins by destroying all the buildings of the opponent. The game also contains a number of neutral units called *sheep.* These are small indestructible units moving randomly around the map making pathfinding and collision detection more complex. Both games are part of the annual ORTS tournament organised by the University of Alberta (Buro, 2007a).

6.3 MAPF in a Full RTS Scenario

We have implemented a MAPF based bot for playing the Full RTS game in ORTS following the proposed steps. Since this work extends previous research on MAPF based bots (and the space limitations prevents us from describing everything in detail), we will concentrate this on the additions we have made. For the details about the MAPF methodology and the Tankbattle scenario, we refer to Papers I and III.

6.3.1 Identifying objects

We identify the following objects in our application: *Workers, Marines, Tanks, Control centers, Barracks, Factories, Cliffs,* and the neutral *Sheep,* and *Minerals.* Units and buildings are present on both sides.

6.3.2 Identifying fields

In the Tankbattle scenario we identified four tasks: Avoid colliding with moving objects, Hunt down the enemy's forces, Avoid colliding with cliffs, and Defend the bases. In the Full RTS scenario we identify the following additional tasks: Mine resources, Create buildings, Train workers and marines, Construct tanks, and Explore the game world. The tasks are organised into the following types of potential fields:

Field of Navigation. This field contains all objects that have an impact on the navigation in the game world: *terrain, own units and buildings, minerals* and *sheep.* The fields are repelling to avoid that our agents collide with the obstacles.

Strategic Field. This field contains the goals for our agents and is an attractive field, different for each agent type. Tanks have attractive fields generated by opponent units and buildings. Workers mining resources have attractive fields generated by mineral patches (or if they cannot carry anymore, the control center where it can drop them off).

Field of Exploration. This field is used by workers assigned to explore the game world and attract them to unexplored areas.

Tactical field. The purpose of the tactical field is to coordinate movements between our agents. This is done by placing a temporary small repelling field at the next movement position for an agent. This prevents own units from moving to the same location if there are other routes available.

Field of spatial planning. This field helps us finding suitable places on the map to construct new buildings such as control centers, barracks and factories at.

This approach has similarities with the work by Paul Tozour in (Tozour, 2004), where the author describes multiple layers of influence maps. Each layer is responsible for handling one task, for example the distance to static objects or the line-of-fire of own agents. The different fields sum up to form a total field that is used as a guide for the agents when selecting actions.

6.3.3 Assigning charges and granularity

Each game object that has an effect on navigation or tactics for our agents has a set of charges which generate a potential field around the center of the object. All fields generated by objects are weighted and summed to form a total field which is used by agents when selecting actions. The initial set of charges was hand crafted. However, the order of importance between the objects simplifies the process of finding good values and the method seems robust enough to allow the bot to work good anyhow. Below is a detailed description of each field. As in the Tankbattle scenario described in (see Paper III), we use a granularity of 1x1 game world points for potential fields, and all dynamic fields are updated every frame.

The opponent units.

Opponent units, tanks marines and workers, generate different fields depending on the agent type and its internal state. In the case of own attacking units, tanks and marines, the opponent units generate attracting symmetric surrounding fields where the highest potentials are at radius equal to the maximum shooting distance, MSD from the enemy unit. This is illustrated in Figure 6.1. It shows a tank (black circle) moving to attack an opponent unit E. The highest potentials (light grey areas) are located in a circle around E.

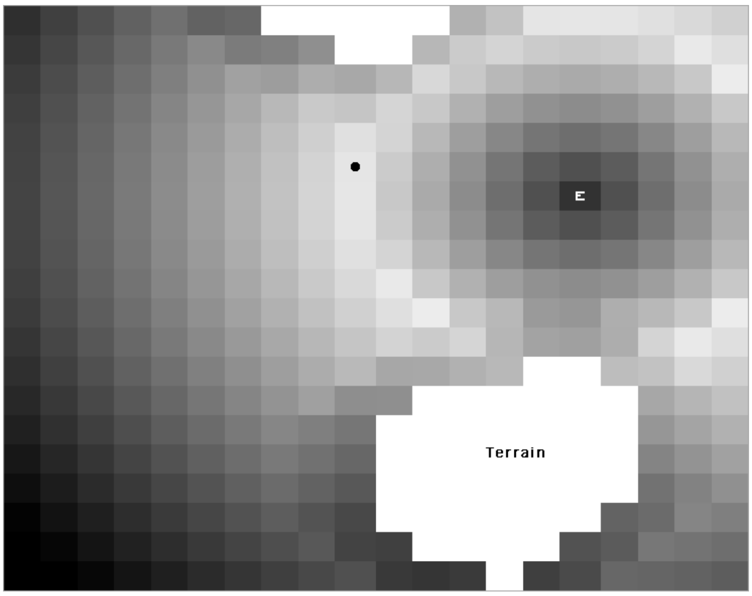


Figure 6.1: *A tank (black circle) engaging an opponent unit E. Light grey areas have higher potential than darker grey areas.*

After an attacking unit has fired its weapon the unit enters a cooldown period when it cannot attack. This cooldown period may be used to retreat from enemy fire, which has shown in Paper III to be a successful strategy. In this case the opponent units generate repelling fields with radius slightly larger than the MSD. The use of a defensive field makes our agents surround the opponent unit cluster at MSD even if the opponent units pushes our agents backwards. This is illustrated in Figure 6.2. The opponent unit E is now surrounded by a strong repelling field that makes the tank (white circle) retreat outside MSD of the opponent.

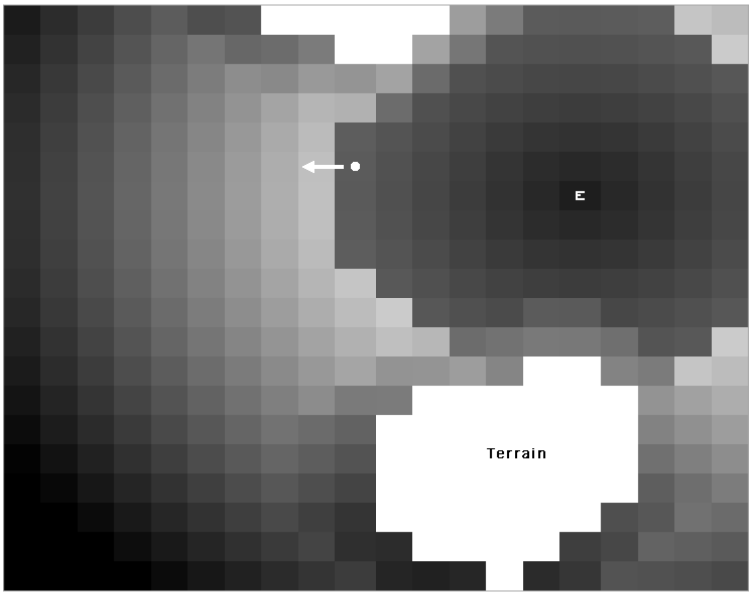


Figure 6.2: *A tank (white circle) in cooldown retreats outside the MSD of an opponent unit.*

The fields generated by game objects are different for different types of own units. In Figure 6.1 a tank is approaching an enemy unit. A tank typically has longer fire range than for example a marine. If a marine would approach the enemy unit a field where the highest potentials are closer to the enemy unit would be generated. Below is pseudocode for calculating the potential an enemy object e generates in a point p in the game world.

```
$distance = \text{distanceBetween(Position } p, \text{EnemyObject } e);$
$potential = \text{calculatePotential}(distance,$
        $\text{OwnObjectType } ot, \text{EnemyObjectType } et);$
```

Own buildings.

Own buildings, control centers barracks and factories, generate repelling fields for obstacle avoidance. An exception is in the case of workers returning minerals to a control center. In this case control centers generate an attractive field calculated using Equation 6.2. The repelling potential $p_{ownB}(d)$ at distance d from the center of the building is calculated using Equation 6.1.

$$p_{ownB}(d) = \begin{cases} 6 \cdot d - 258 & \text{if } d <= 43 \\ 0 & \text{if } d > 43 \end{cases} \tag{6.1}$$

$$p_{attractive}(d) = \begin{cases} 240 - d \cdot 0.32 & \text{if } d <= 750 \\ 0 & \text{if } d > 750 \end{cases} \tag{6.2}$$

Minerals.

Minerals generate two different types of field; one attractive field used by workers mining resources and a repelling field that is used for obstacle avoidance. The potential $p_{attractive}(d)$ at distance d from the center of a mineral is calculated using Equation 6.2. In the case when minerals generate a repelling field, the potential $p_{mineral}(d)$ at distance d from the mineral is calculated as:

$$p_{mineral}(d) = \begin{cases} -20 & \text{if } d <= 8 \\ 20 - 2 \cdot d & \text{if } d \in]8, 10] \end{cases} \tag{6.3}$$

Figure 6.3 and 6.4 illustrates a worker mining resources from a nearby mine. In Figure 6.3 the worker is ordered to gather more resources and an attractive potential field is placed around the mine. Terrain, own worker units and the base all generate small repelling fields used for obstacle avoidance. When the worker has gathered as much resources it can carry, it must return to the base to drop them off. This is shown in Figure 6.4. The attractive charge is now placed in the center of the base, and the mine now generates a small repelling field for obstacle avoidance.

Field of exploration.

The field of exploration is a field with attractive charges at the positions in the game world that need to be explored. First an importance value for each terrain tile is calculated in order to find next position to explore. This process is described below. Once a position is found, the Field of Navigation, Equation 6.4, is used to guide the unit to the spot. This approach seems to be more robust than letting all unexplored areas generate attractive potentials. In the latter case explorer units tend to get stuck somewhere in the middle of the map due to the attractive potentials generated from unexplored areas in several directions.

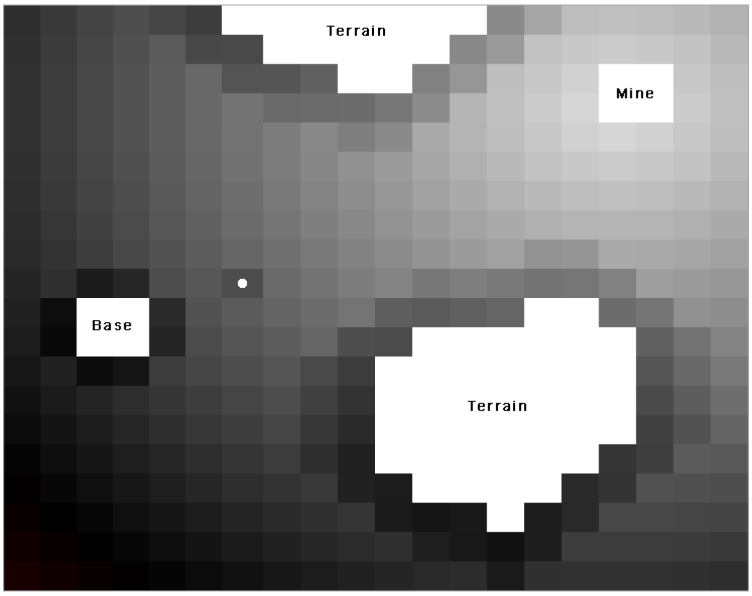


Figure 6.3: *A worker unit (white circle) moving towards a mine to gather resources. The mine generates an attractive field and mountains (black) generate small repelling fields for obstacle avoidance. Light grey areas are more attracting than darker grey areas.*

$$p_{navigation}(d) = \begin{cases} 150 - d * 0.1 & \text{if } d <= 1500 \\ 0 & \text{if } d > 1500 \end{cases} \tag{6.4}$$

The importance value for each tile is calculated as follows:

I. Each terrain tile (16x16 points) is assigned an explore value, $E(x, y)$, initially set to 0.

II. In each frame, $E(x, y)$ is increased by 1 for all passable tiles.

III. If a tile is visible by one or more of our own units in the current frame, its $E(x, y)$ is reset to 0.

IV. Calculate an importance value for each tile using Equation 6.5. The distance d is the distance from the explorer unit to the tile.

$$importance(x, y, d) = 2.4 \cdot E(x, y) - 0.1d \tag{6.5}$$

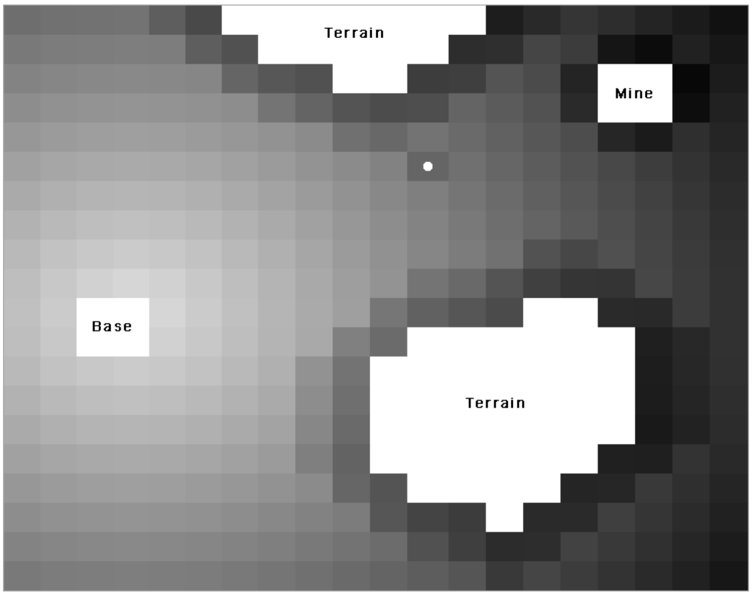


Figure 6.4: *A worker unit (white circle) moving towards a base to drop of gathered resources.*

Figure 6.5 illustrates a map with a base and an own explorer unit. The white areas of the map are unexplored, and the areas visible by own units or buildings are black. The grey areas are previously explored areas that currently are not visible by own units or buildings. Light grey tiles have higher explore values than darker grey tiles.

The next step is to pick the tile of the greatest importance (if there are several equally important, pick one of them randomly), and let it generate the field. This is shown in Figure 6.6. The explorer unit move towards the choosen tile from Figure 6.5 to explore next.

Base building.

When a worker is assigned to construct a new building, a suitable build location must first be found. The method used to find the location is described in the SpatialPlanner agent section below. Once a location is found, the potential $p_{builder}(d)$ at distance d from the position to build at is calculated using the Field of Navigation (see Equation 6.4).

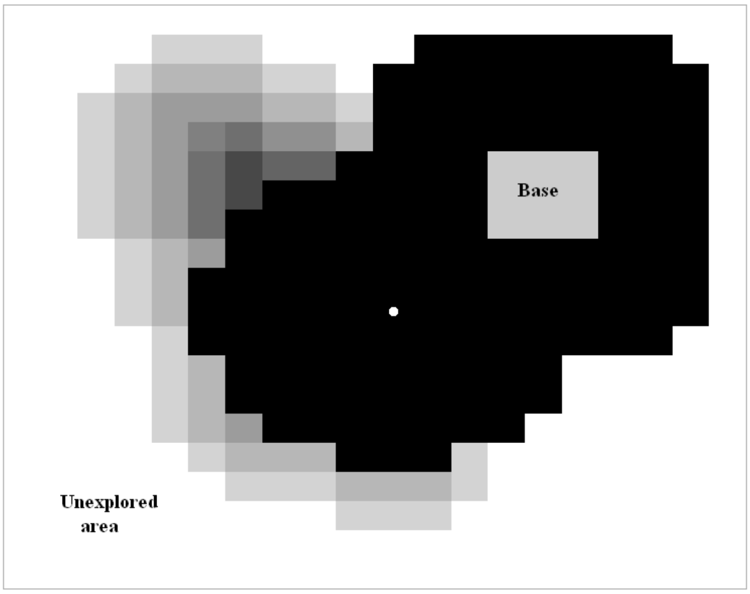


Figure 6.5: *Explore values as seen by the explorer unit (white circle). Grey areas have previously been visited. Black areas are currently visible by an own unit or building.*

6.3.4 The agents of the bot

Each own unit (worker, marine or tank) is represented by an agent in the system. The multi-agent system also contains a number of agents not directly associated with a physical object in the game world. The purpose of these agents is to coordinate own units to work towards common goals (when applicable) rather than letting them act independently. Below follows a more detailed description of each agent.

CommanderInChief.

The CommanderInChief agent is responsible for making an overall plan for the game, called a battleplan. The battleplan contains the order of creating units and buildings, for example start with training 5 workers then build a barrack. It also contains special actions, for example sending units to explore the game world. When one post in the battleplan is completed, the next one is executed. If a previously completed post no longer is satisfied, for example a worker is killed or a barrack is destroyed, the CommanderInChief agent takes the necessary actions for completing that post before resuming current actions. For a new post to be executed there must be enough resources available.

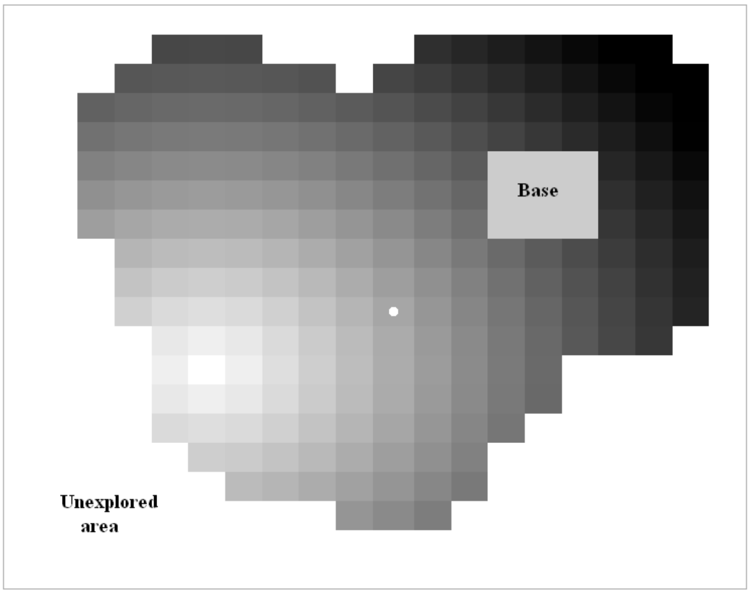


Figure 6.6: *The explorer unit (white circle) move towards the tile with the highest importance value (light grey area).*

The battleplan is based on the ideas of subsumption architectures (see (Brooks, 1986)) shown in Figure 6.7. Note that all workers, unless ordered to do something else, are gathering resources.

CommanderInField.

The CommanderInField agent is responsible for executing the battleplan generated by the CommanderInChief. It sets the goals for each unit agent, and change goals during the game if necessary (for example use a worker agent currently gathering resources to construct a new building, and to have the worker go back to resource gathering after the building is finished). The CommanderInField agent has three additional agents to help it with the execution of the battleplan; GlobalMapAgent, AttackCoordinator and SpatialPlanner.

GlobalMapAgent.

In ORTS a data structure with the current game world state is sent each frame from the server to the connected clients. The location of buildings are however only included

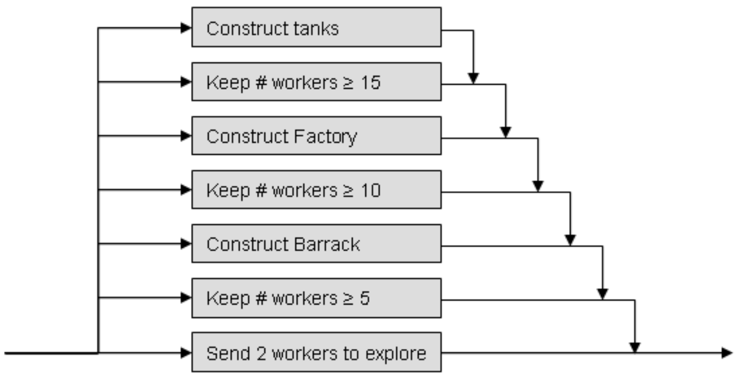


Figure 6.7: *The subsumption hierarchy battleplan.*

in the data structure if an own unit is within visibility range of the building. It means that an enemy base that has been spotted by an own unit and that unit is destroyed, the location of the base is no longer sent in the data structure. Therefore our bot has a dedicated global map agent to which all detected objects are reported. This agent always remembers the location of previously spotted enemy bases until a base is destroyed, as well as distributes the positions of detected enemy units to all the own units.

AttackCoordinator.

The purpose of the attack coordinator agent is to optimize attacks at enemy units. The difference between using the coordinator agent compared to attacking the most damaged unit within fire range (which seemed to be the most common approach used in the 2007 years' ORTS tournament) is best illustrated with an example. A more detailed description of the attack coordinator can be found in Paper I.

In Figure 6.8 the own units A, B and C deals 3 damage each. They can all attack opponent unit X (X can take 8 more damage before it is destroyed) and unit A can also attack unit Y (Y can take 4 more damage before it is destroyed). If an *attack the weakest* strategy is used, unit A will attack Y, and B and C will attack X with the result that both X and Y will survive. By letting the coordinator agent optimize the attacks, all units are coordinated to attack X, which then is destroyed and only Y will survive.

SpatialPlanner.

To find a suitable location to construct new buildings at, we use a special type of field only used to find a spot to build at. Once it has been found by the Spatial Planning Agent,

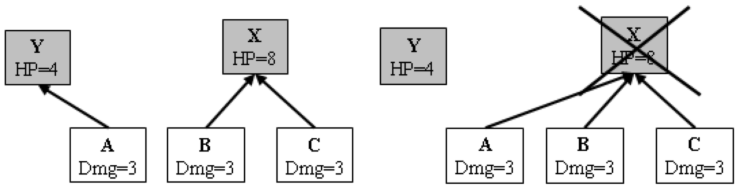


Figure 6.8: *Attacking the most damaged unit (to the left) vs. Optimize attacks (to the right).*

a worker agent uses the Field of Navigation (see Equation 6.4) to move to that spot. Below follow equations used to calculate the potential game objects generate in the spatial planning field. *Own buildings*. Own buildings generate a field with an inner repelling area (to avoid construct buildings too close to each other) and an outer attractive area (for buildings to be grouped together). Even though the size differs somewhat between buildings for simplicity we use the same formula regardless of the type of building. The $p_{ownbuildings}(d)$ at distance d from the center of an own building is calculated as:

$$p_{ownbuildings}(d) = \begin{cases} -1000 & \text{if } d <= 115 \\ 230 - d & \text{if } d \in]115, 230] \\ 0 & \text{if } d > 230 \end{cases} \tag{6.6}$$

Enemy buildings. Enemy buildings generate a repelling field. The reason is of course that we do not want own buildings to be located too close to the enemy. The $p_{enemybuildings}(d)$ at distance d from the center of an enemy building is calculated as:

$$p_{enemybuildings}(d) = \begin{cases} -1000 & \text{if } d <= 150 \\ 0 & \text{if } d > 150 \end{cases} \tag{6.7}$$

Minerals. It is not possible to construct buildings on top of minerals therefore they have to generate repelling fields. The $p_{mineral}(d)$ at distance d from the center of a mineral is calculated using Equation 6.8. The field is slightly attractive outside the repelling area since it is beneficial to have bases located close to resources.

$$p_{mineral}(d) = \begin{cases} -1000 & \text{if } d <= 90 \\ 5 - d \cdot 0.02 & \text{if } d \in]90, 250] \\ 0 & \text{if } d > 250 \end{cases} \tag{6.8}$$

Impassable terrain. Cliffs generate a repelling field to avoid workers trying to construct a building too close to a cliff. The $p_{cliff}(d)$ at distance d from the closest cliff is calculated as:

$$p_{cliff}(d) = \begin{cases} -1000 & \text{if } d <= 125 \\ 0 & \text{if } d > 125 \end{cases} \tag{6.9}$$

Game world edges. The edges of the game world have to be repelling as well to avoid workers trying to construct a building outside the map. The $p_{edge}(d)$ at distance d from the closest edge is calculated as:

$$p_{edge}(d) = \begin{cases} -1000 & \text{if } d < 90 \\ 0 & \text{if } d >= 90 \end{cases} \tag{6.10}$$

To find a suitable location to construct a building at, we start by calculating the total buildspot potential in the current position of the assigned worker unit. In the next iteration we calculate the buildspot potential in points at a distance of 4 tiles from the location of the worker, in next step at distance 8, and continue up to distance 200. The position with the highest buildspot potential is the location to construct the building at. Figure 6.9 illustrates the field used by the Spatial Planner Agent to find a spot for the worker (black circle) to construct a new building at. Lighter grey areas are more attractive than darker grey areas. The location to construct the building at is shown as a black non-filled rectangle. Once the spot is found the worker agent uses the Field of Navigation to move to that location.

6.4 Experiments

We used the ORTS tournament of 2008 as a benchmark to test the strength of our bot. The number of participants in the Full RTS game was unfortunately very low, but the results are interesting anyway since the opponent team from University of Alberta has been very competitive in earlier tournaments. The UOFA bot uses a hierarchy of commanders where each major task such as gathering resources or building a base is controlled by a dedicated commander. The Attack commander, responsible for hunting down and destroy enemy forces, gather units in squads and uses A* for the pathfinding. The results from the tournament are shown in Table 6.1. Our bot won 82.5% of the games against the opponent team over 2x200 games (200 different maps where the players switched sides).

6.5 Discussion

There are several interesting aspects here.

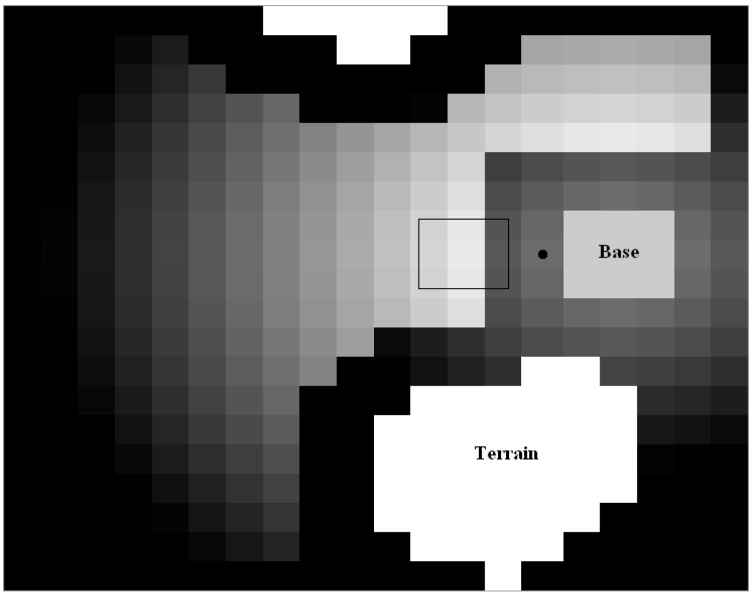


Figure 6.9: *Field used by he Spatial Planner agent to find a build spot (black non-filled rectangle).*

- First, we show that the approach we have taken, to use Multi-agent potential fields, is a viable way to construct highly competitive bots for RTS scenarios of medium complexity. Even though the number of competitors this year was very low, the opponent was the winner (89 % wins) of the 2007 tournament. Unfortunately, ORTS server updates have prevented us from testing our bot against the other participant of that year, but there are reasons to believe that it would manage well against those solutions too (although it is not sure, since the winning relation between strategies in games is not transitive, see e.g. Rock, Paper Scissors (deJong, 2004)). We argue though that the use of open tournaments as a benchmark is still better than if we constructed the opponent bots ourselves.

- Second, we combine the ideas of using a role-oriented MAS architecture and MAPF bots.

- Third, we introduce (using the potential field paradigm) a way to place new buildings in RTS games.

Team	Win %	Wins/games	DC
Blekinge	82.5%	(330/400)	0
Uofa	17.5%	(70/400)	3

Table 6.1: *Results from the* ORTS *tournament of 2008. DC is the number of disconnections due to client software failures.*

6.6 Conclusions and Future Work

We have constructed an ORTS bot based on both the principles of role-oriented MAS and Multi-agent Potential Fields. The bot is able to play an RTS game and outperforms the competitor by winning more than 80% of the games in an open tournament where it participated.

Future work will include to generate a battleplan for each game depending on the skill and the type of the opponent it is facing. The strategy of our bot is now fixed to construct as many tanks as possible to win by brute strength. It can quite easily be defeated by attacking our base with a small force of marines before we are able to produce enough tanks. The CommanderInChief agent should also be able to change battleplan to adapt to changes in the game to, for example, try to recover from an attack by a marine force early in the game. Our bot is also set to always directly rebuild a destroyed building. If, for example, an own factory is destroyed it might not be the best option to directly construct a new one. It might be better to train marines and/or move attacking units back to the base to get rid of the enemy units before constructing a new factory. There are also several other interesting techniques to replace the sum-sumption architecture.

We believe that a number of details in the higher level commander agents may improve in the future versions when we better adapt to the opponents. We do however need more opponent bots that uses different strategies to improve the validation.

CHAPTER

SEVEN

PAPER VI

A Multi-agent Potential Fields based bot for Real-time Strategy Games.

Johan Hagelbäck & Stefan J. Johansson

International Journal of Computer Games Technology, vol. 2009, Article ID 910819, 10 pages. doi:10.1155/2009/910819.

7.1 Introduction

A *Real-time Strategy (*RTS*)* game is a game in which the players use resource gathering, base building, technological development and unit control in order to defeat its opponent(s), typically in some kind of war setting. The RTS game is not turn-based in contrast to board games such as Risk and Diplomacy. Instead, all decisions by all players have to be made in real-time. Generally the player has a top-down perspective on the battlefield although some 3D RTS games allow different camera angles. The real-time aspect makes the RTS genre suitable for multiplayer games since it allows players to interact with the game independently of each other and does not let them wait for someone else to finish a turn.

In RTS games computer bots often "cheats", i.e. they have complete visibility (perfect information) of the whole game world. The purpose is to have as much information available as possible for the AI to reason about tactics and strategies in a certain environment. Cheating is, according to Nareyek, *"very annoying for the player if discovered"* and he predicts the game AIs to get a larger share of the processing power in the future which in turn may open up for the possibility to use more sophisticated AIs (Nareyek, 2004). The human player in most modern RTS games does not have this luxury, instead the player only has visibility of the area populated by the own units and the rest of the game world is unknown until it gets explored. This property of incomplete information is usually referred to as *Fog of War* or FoW.

In 1985 Ossama Khatib introduced a new concept while he was looking for a real-time obstacle avoidance approach for manipulators and mobile robots. The technique which he called *Artificial Potential Fields* moves a manipulator in a field of forces. The position to be reached is an attractive pole for the end effector (e.g. a robot) and obstacles are repulsive surfaces for the manipulator parts (Khatib, 1986). Later on Arkin (Arkin, 1987) updated the knowledge by creating another technique using superposition of spatial vector fields in order to generate behaviours in his so called motor schema concept.

Many studies concerning potential fields are related to spatial navigation and obstacle avoidance, see e.g. (Borenstein & Koren, 1991; Massari et al., 2004). The technique is really helpful for the avoidance of simple obstacles even though they are numerous. Combined with an autonomous navigation approach, the result is even better, being able to surpass highly complicated obstacles (Borenstein & Koren, 1989).

Lately some other interesting applications for potential fields have been presented. The use of potential fields in architectures of multi agent systems is giving quite good results defining the way of how the agents interact. Howard et al. developed a mobile sensor network deployment using potential fields (Howard et al., 2002), and potential fields have been used in robot soccer (Johansson & Saffiotti, 2002; Röfer et al., 2004). Thurau et al. (Thurau et al., 2004b) has developed a game bot which learns reactive behaviours (or potential fields) for actions in the First-Person Shooter game Quake II through imitation.

The article is organised as follows. First, we propose a methodology for Multi-agent Potential Fields (MAPF) based solution in a RTS game environment. We will show how the methodology can be used to create a bot for a resource gathering scenario (Section 7.4) followed by a more complex tankbattle scenario in Section 7.5. We will also present some preliminary results on how to deal with imperfect information, fog of war (Section 7.6). The methodology has been presented in the previous Papers I and III. This article summarises the previous work and extends it by adding new experiments and new results. Last in this article we have a discussion and line out some directions for future work.

7.2 A Methodology for Multi-agent Potential Fields

When constructing a multi-agent potential fields based system for controlling agents in a certain domain, there are a number of issues that we must take in consideration. It is for example important that each interesting object in the game world generates some type of field, and we must decide which objects can use static fields to decrease computation time.

To structure this, we identify six phases in the design of a MAPF-based solution:

I. The identification of objects,

II. The identification of the driving forces (i.e. the fields) of the game,

III. The process of assigning charges to the objects,

IV. The granularity of time and space in the environment,

V. The agents of the system, and

VI. The architecture of the MAS.

In *the first phase*, we may ask us the following questions: What are the *static objects* of the environment? That is: what objects remain their attributes throughout the lifetime of the scenario? What are the *dynamic objects* of the environment? Here we may identify a number of different ways that objects may change. They may move around, if the environment has a notion of physical space. They may change their attractive (or repulsive) impact on the agents. What is the *modifiability* of the objects? Some objects may be consumed, created, or changed by the agents.

In *the second phase*, we identify the driving forces of the game at a rather abstract level, e.g. to avoid obstacles, or to base the movements on what the opponent does. This leads us to a number of fields. The main reason to enable multiple fields is that it is very easy to isolate certain aspects of the computation of the potentials if we are able to filter out a certain aspect of the overall potential, e.g. the repulsive forces generated by the terrain in a physical environment. We may also dynamically weight fields separately, e.g. in order to decrease the importance of the navigation field when a robot stands still in a surveillance mission (and only moves its camera). We may also have *strategic fields* telling the agents in what direction their next goal is, or *tactical fields* coordinating the movements with those of the teammate agents.

The third phase includes placing the objects in the different fields. Static objects should typically be in the *field of navigation*. The potentials of such a field are precalculated in order to save precious run time CPU resources.

In *the fourth phase*, we have to decide the resolution of space and time. If the agents are able to move around in the environment, both these measures have an impact on the lookahead. The space resolution obviously, since it decides what points in space that we are able to access, and the time in that it determines how far we may get in one time frame (before it is time to make the next decision about what to do).

The fifth phase is to decide what objects to agentify and set the repertoire of those agents: what actions are we going to evaluate in the lookahead? As an example, if the agent is omnidirectional in its movements, we may not want to evaluate all possible points that the agent may move to, but rather try to filter out the most promising ones by using some heuristic, or use some representable sample.

In *the sixth step*, we design the architecture of the MAS. Here we take the unit agents identified in the fifth phase, give them roles and add the supplementary agents (possibly) needed for coordination, and special missions (not covered by the unit agents themselves).

7.3 ORTS

Open Real Time Strategy (ORTS) (Buro, 2007a) is a real-time strategy game engine developed as a tool for researchers within artificial intelligence (AI) in general and game AI in particular. ORTS uses a client-server architecture with a game server and players connected as clients. Each timeframe clients receive a data structure from the server containing the current game state. Clients can then call commands that activate and control their units. Commands can be like move unit A to (x, y) or attack opponent unit X with unit A. The game server executes the client commands in random order.

Users can define different type of games in scripts where units, structures and their interactions are described. All type of games from resource gathering to full real time strategy (RTS) games are supported.

We will begin by looking at a one-player resource gathering scenario game called *Collaborative Pathfinding*, which was part of the 2007 and 2008 ORTS competitions (Buro, 2007a). In this game the player has 20 worker units. The goal is to use the workers to mine resources from nearby mineral patches and return them to a base. A worker must be adjacent to a mineral object to mine, and to a base to return resources. As many resources as possible shall be collected within 10 minutes.

This is followed by looking at the two-player games, *Tankbattle*, which was part of the 2007 and 2008 ORTS competitions (Buro, 2007a) as well.

In *Tankbattle* each player has 50 tanks and five bases. The goal is to destroy the bases of the opponent. Tanks are heavy units with long fire range and devastating firepower but a long cool-down period, i.e. the time after an attack before the unit is ready to attack again. Bases can take a lot of damage before they are destroyed, but they have no defence mechanism of their own so it may be important to defend own bases with tanks. The map in a tankbattle game has randomly generated terrain with passable lowland and impassable cliffs.

Both games contain a number of neutral units (sheep). These are small indestructible units moving randomly around the map. The purpose of sheep is to make pathfinding and collision detection more complex.

7.4 Multi-agent Potential Fields in ORTS

First we will describe a bot playing the Collaborative Pathfinding game based on MAPF following the proposed methodology. Collaborative Pathfinding is a 1-player game where the player has one control center and 20 worker units. The aim is to move workers to mineral patches, mine up to 10 resources (the maximum load a worker can carry), then return to a friendly control center to drop them off.

7.4.1 Identifying objects

We identify the following objects in our application: *Cliffs, Sheep, Base stations* and *workers.*

7.4.2 Identifying fields

We identified five tasks in ORTS: Avoid colliding with the terrain, Avoid getting stuck at other moving objects, Avoid colliding with the bases, Move to the bases to leave resources and Move to the mineral patches to get new resources. This leads us to three major types of potential fields: A *Field of Navigation*, a *Strategic Field* and a *Tactical field*.

The field of navigation is a field generated by repelling static terrain. This is because we would like the agents to avoid getting too close to objects where they may get stuck, but instead smoothly passing around them.

The strategic field is a dynamic attracting field. It makes agents go towards the mineral patches to mine, and return to the base to drop off resources.

Own workers, bases and sheep generate small repelling fields. The purpose of these fields is the same as for obstacle avoidance; we would like our agents to avoid colliding with each other and bases as well as avoiding the sheep. This task is managed by the tactical field.

7.4.3 Assigning charges

Each worker, base, sheep and cliffs has a set of charges which generates a potential field around the object. These fields are weighted and summed together to form a total potential field that is used by our agents for navigation.

Cliffs, e.g. impassable terrain, generate a repelling field for obstacle avoidance. The field is constructed by copying pre-generated matrixes of potentials into the field of navigation when a new game is started. The potential all cliffs generate in a point (x, y) is calculated as the lowest potential a cliff generates in that point. The potential $p_{cliff}(d)$

in a point at distance d from the closest impassable terrain tile is calculated as:

$$p_{cliff}(d) = \begin{cases} -80/(d/8)^2 & \text{if } d > 0 \\ -80 & \text{if } d = 0 \end{cases} \tag{7.1}$$

Own worker units generate repelling fields for obstacle avoidance. The potential $p_{worker}(d)$ at distance d from the center of another worker is calculated as:

$$p_{worker}(d) = 1.429 \cdot \begin{cases} -20 & \text{if } d <= 6 \\ 16 - 2 \cdot d & \text{if } d \in]6, 8] \end{cases} \tag{7.2}$$

Sheep. Sheep generate a small repelling field for obstacle avoidance. The potential $p_{sheep}(d)$ at distance d from the center of a sheep is calculated as:

$$p_{sheep}(d) = 0.125 \cdot \begin{cases} -20 & \text{if } d <= 8 \\ 2 \cdot d - 25 & \text{if } d \in]8, 12.5] \end{cases} \tag{7.3}$$

Own bases. The own bases generate two different fields depending on the current state of a worker. The base generates an attractive field if the worker need to move to the base and drop off its resources. Once it has arrived at the base all the resources are dropped. The potential $p_{attractive}(d)$ at distance d from the center of the base is calculated as:

$$p_{attractive}(d) = \begin{cases} 240 - d \cdot 0.32 & \text{if } d <= 750 \\ 0 & \text{if } d > 750 \end{cases} \tag{7.4}$$

In all other states of the worker, the own base generates a repelling field for obstacle avoidance. Below is the function for calculating the potential $p_{ownB}(d)$ at distance d from the center of the base. Note that this is, of course, the view of the worker. The base will effect some of the workers with the attracting field while at the same time effect the rest with an repelling field. If a point is inside the quadratic area the base occupies, the potential in that points is always -10000 (potential used for impassable points).

$$p_{ownB}(d) = 0.125 \cdot \begin{cases} 6 \cdot d - 258 & \text{if } d <= 43 \\ 0 & \text{if } d > 43 \end{cases} \tag{7.5}$$

Minerals, similar to own bases, generate attractive fields for all workers that do not carry maximum loads and a repelling field for obstacle avoidance when they do. The

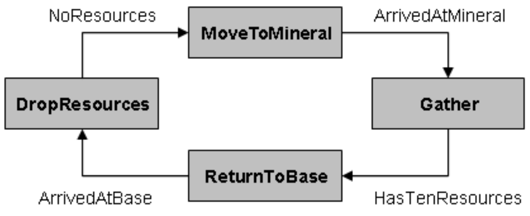


Figure 7.1: *The finite state machine used by the workers in a resource gathering scenario.*

potential of the attractive field is the same as the attractive field around the own base in Equation 7.4.

In the case when minerals generate a repelling field, the potential $p_{mineral}(d)$ at distance d from the center of a mineral is calculated as:

$$p_{mineral}(d) = 1.429 \cdot \begin{cases} -20 & \text{if } d <= 8 \\ 20 - 2 \cdot d & \text{if } d \in]8, 10] \end{cases} \tag{7.6}$$

7.4.4 The Granularity of the System

Since the application is rather simple, we use full resolution of both the map and the time frames without any problems.

7.4.5 The agents

The main units of our system are the workers. They use a simple finite state machine (FSM) illustrated in Figure 7.1 to decide what state they are in (and thus what fields to activate). No central control or explicit coordination is needed, since the coordination is emerging through the use of the charges.

7.4.6 The Multi-Agent System Architecture

In addition to the worker agents, we have one additional agent that is the interface between the workers, and the game server. It receives server information about the positions of all objects and workers which it distributes to the worker agents. They then decide what to do, and submit their proposed actions to the interface agent which in turn sends them through to the ORTS server.

Team	Matches	AvgResources	Disconnected
BTH	250	5630.72	30
Uofa	250	4839.6	0

Table 7.1: *Experiment results from the Collaborative Pathfinding game in 2008 years' tournament.*

7.4.7 Experiments, resource gathering

Table 7.1 shows the result from the Collaborative Pathfinding game in 2008 years' ORTS tournament. It shows that a MAPF based bot can compete with A* based solutions in a resource gathering scenario. There are however some uncertainties in these results. Our bot has disconnected from the server (i.e. crashed) in 30 games. The reason for this is not yet clear and must be investigated in more detail. Another issue is that Uofa has used the same bot that they used in the 2007 years' tournament, and the bot had a lower score this year. The reason, according to the authors, was "probably caused by a pathfinding bug we introduced" (Buro, 2008). Still we believe that with some more tuning and bug fixing our bot can probably match the best bots in this scenario.

7.5 MAPF in ORTS, Tankbattle

In the 2-player Tankbattle game each player has a number of tanks and bases, and the goal is to destroy the opponent bases. In Paper I we describe the implementation of an ORTS bot playing Tankbattle based on MAPF following the proposed methodology. This bot was further improved in Paper III where a number of weaknesses of the original bot were addressed. We will now, just as in the case of the Collaborative pathfinding scenario, present the six steps of the used methodology. However, there are details in the implementation of several of these steps that we have improved and shown the effect of in experiments. We will therefore, to improve the flow of the presentation, not present all of them in chronologic order. Instead we start by presenting the ones that we have kept untouched through the series of experiments.

7.5.1 Identifying objects

We identify the following objects in our application: Cliffs, Sheep, and own (and opponent) tanks and base stations.

7.5.2 Identifying fields

We identified four tasks in ORTS: *Avoid colliding with the terrain, Avoid getting stuck at other moving objects, Hunt down the enemy's forces* and *Defend the bases*. In the re-

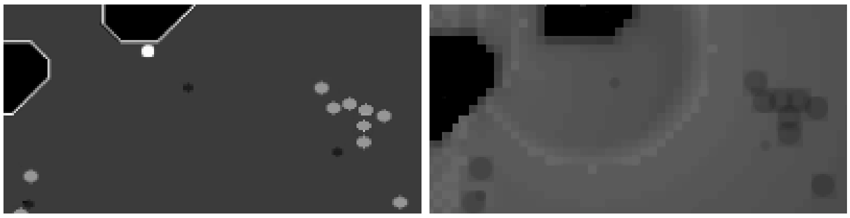


Figure 7.2: *Part of the map during a tankbattle game. The left picture shows our agents (light-grey circles), an opponent unit (white circle) and three sheep (small dark-grey circles). The right picture shows the total potential field for the same area. Light areas have high potential and dark areas low potential.*

source gathering scenario we used the two major types *Field of Navigation* and *Strategic Field*. Here we add a new major type of potential field: the *Defensive Field*.

The field of navigation is, as in the previous example of Collaborative pathfinding, a field generated by repelling static terrain for obstacle avoidance.

The strategic field is an attracting field. It makes units go towards the opponents and place themselves on appropriate distances where they can fight the enemies.

The defensive field is a repelling field. The purpose is to make own agents retreat from enemy tanks when they are in cooldown phase. After an agent has attacked an enemy unit or base, it has a cooldown period when it cannot attack and it is therefore a good idea to stay outside enemy fire range while in this phase. The defensive field is an improvement to deal with a weakness found in the original bot described in Paper I.

Own units, own bases and sheep generate small repelling fields. The purpose is the same as for obstacle avoidance; we would like our agents to avoid colliding with each other or bases as well as avoiding the sheep. This is managed by the tactical field.

7.5.3 Assigning charges

The upper picture in Figure 7.2 shows part of the map during a tankbattle game. The screenshot are from the 2D GUI available in the ORTS server. It shows our agents (light-grey circles) moving in to attack an opponent unit (white circle). The area also has some cliffs (black areas) and three sheep (small dark-grey circles). The lower picture shows the total potential field in the same area. Dark areas have low potential and light areas high potential. The light ring around the opponent unit, located at maximum shooting distance of our tanks, is the distance our agents prefer to attack opponent units from. The picture also shows the small repelling fields generated by our own agents and the sheep.

Cliffs. Cliffs generate the same field as in the resource gathering scenario, see 7.4.3.

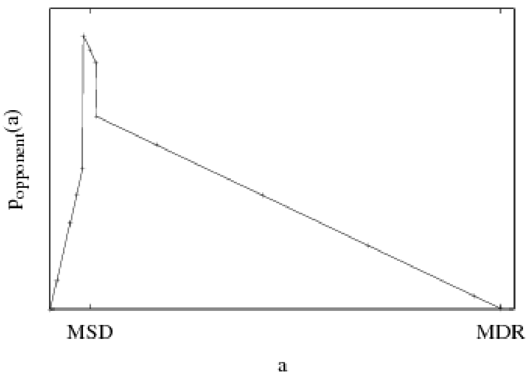


Figure 7.3: *The potential $p_{opponent}(a)$ generated by opponent units as a function of the distance a.*

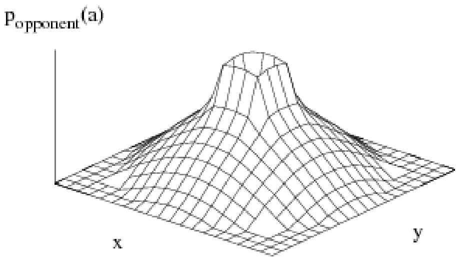


Figure 7.4: *The potential $p_{opponent}(a)$ generated by the opponent that is in the middle.*

The opponent units and bases. All opponent units and bases generate symmetric surrounding fields where the highest potential is in a ring around the object with a radius of MSD (*Maximum Shooting Distance*). MDR refers to the *Maximum Detection Range*, the distance from which an agent starts to detect the opponent unit. The reason why the location of the enemy unit is not the final goal is that we would like our units to surround the enemy units by attacking from the largest possible distance. The potential all opponent units generate in a certain point is then equal to the highest potential *any* opponent unit generates in that point, and not the sum of the potentials that all opponent units generate. If we were to sum the potentials, the highest potential and most attractive destination would be in the center of the opponent unit cluster. This was the case in the first version of our bot and was identified as one of its major weaknesses. The potentials $p_{oppU}(d)$ and $p_{oppB}(d)$ at distance d from the center of an agent and with $D =$MSD and $R =$ MDR are calculated as:[1]

$$p_{oppU}(d) = 0.125 \cdot \begin{cases} 240/d(D-2), & \text{if } d \in [0, D-2[\\ 240, & \text{if } d \in [D-2, D] \\ 240 - 0.24(d-D) & \text{if } d \in]D, R] \end{cases} \tag{7.7}$$

$$p_{oppB}(d) = 0.125 \cdot \begin{cases} 360/d(D-2), & \text{if } d \in [0, D-2[\\ 360, & \text{if } d \in [D-2, D] \\ 360 - 0.32(d-D) & \text{if } d \in]D, R] \end{cases} \tag{7.8}$$

Own units generate repelling fields for obstacle avoidance. The potential $p_{ownU}(d)$ at distance d from the center of a unit is calculated as:

$$p_{ownU}(d) = 0.125 \cdot \begin{cases} -20 & \text{if } d <= 14 \\ 32 - 2 \cdot d & \text{if } d \in]14, 16] \end{cases} \tag{7.9}$$

Own bases generate repelling fields similar to the fields around the own bases described in Section 7.4.3.

Sheep generate the same weak repelling fields as in the Collaborative pathfinding scenario, see Section 7.4.3.

7.5.4 The multi-agent architecture

In addition to the interface agent dealing with the server (which is more or less the same as in the collaborative pathfinding scenario), we use a coordinator agent to globally

[1] $I = [a, b[$ denote the half-open interval where $a \in I$, but $b \notin I$

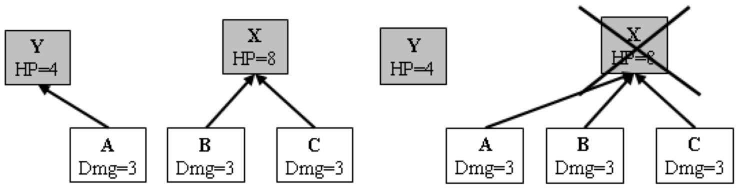


Figure 7.5: *Attacking the most damaged unit (to the left) vs. Optimize attacks (to the right).*

coordinate the attacks on opponent units to maximize the number of opponent units destroyed. The difference between using the coordinator agent compared to attacking the most damaged unit within fire range is best illustrated with an example.

Left in Figure 7.5 the own units A, B and C deals 3 damage each. They can attack opponent unit X (can take 8 more damage before it is destroyed) and unit Y (can take 4 more damage before it is destroyed). Only unit A can attack enemy unit Y. The most common approach in the ORTS tournament (Buro, 2007a) was to attack the most damaged enemy unit within firerange. In the example both enemy unit X and Y would be attacked, but both would survive to answer the attacks.

With the coordinator agent attacks would be spread out as to the right in Figure 7.5. In this case enemy unit X would be destroyed and only unit Y can answer the attacks.

7.5.5 The granularity of the system

Each unit (own or enemy), base, sheep and cliffs has a set of charges which generates a potential field around the object. These fields are weighted and summed together to form a total potential field that is used by our agents for navigation.

In Paper I we used pre-generated fields that were simply added to the total potential field at runtime. To reduce memory and CPU resources needed the game world was split into tiles where each tile was 8x8 points in the game world. This proved not to be detailed enough and our agents often got stuck in terrain and other game objects. The results as shown in Table 7.2 are not very impressive and our bot bot only won 14% of the played games.

Some notes on how the results are presented:

- *Avg units.* This is the average number of units (tanks) our bot had left after a game is finished.

- *Avg bases.* This is the average number of bases our bot had left after a game is finished.

Team	Wins ratio	Wins/games	Avg units	Avg bases	Avg score
NUS	0%	(0/100)	0.01	0.00	-46.99
WarsawB	0%	(0/100)	1.05	0.01	-42.56
UBC	24%	(24/100)	4.66	0.92	-17.41
Uofa.06	32%	(32/100)	4.20	1.45	-16.34
Average	14%	(14/100)	2.48	0.60	-30.83

Table 7.2: *Experiment results from the original bot*

Team	Wins ratio	Wins/games	Avg units	Avg bases	Avg score
NUS	9%	(9/100)	1.18	0.57	-32.89
WarsawB	0%	(0/100)	3.03	0.12	-36.71
UBC	24%	(24/100)	16.11	0.94	0.46
Uofa.06	42%	(42/100)	10.86	2.74	0.30
Average	18.75%	(18.75/100)	7.80	1.09	-17.21

Table 7.3: *Experiment results from increased granularity*

- *Avg score.* This is the average score for our bot after a game is finished. The score is calculated as

$$\begin{aligned} score =& 5(ownBasesLeft - oppBasesLeft) + \\ & ownUnitsLeft - oppUnitsLeft \end{aligned} \tag{7.10}$$

In Paper III we proposed a solution to this problem. Instead of dividing the game world into tiles the resolution of the potential fields was set to 1x1 points. This allows navigation at the most detailed level. To make this computationally feasible we calculate the potentials at runtime, but only for those points that are near own units that are candidates to move to in the next time frame. In total we calculate nine potentials per unit, eight directions and the potential of staying in the position it is. The results, as shown in Table 7.3, show a slight increase in the number of games won and a large improvement in the game score.

7.5.6 Adding an additional field

Defensive field. After a unit has fired its weapon the unit has a cooldown period when it cannot attack. In the original bot our agents were, as long as there were enemies within maximum shooting distance (MSD), stationary until they were ready to fire again. The cooldown period can instead be used for something more useful and in Paper III we proposed the use of a defensive field. This field make the units retreat when they cannot attack and advance when they are ready to attack once again. With this enhancement our agents always aims to be at MSD of the closest opponent unit or base and surrounds the opponent unit cluster at MSD. The potential $p_{defensive}(d)$ at distance d from the center of an agent is calculated using the formula in Equation 7.11.

Team	Wins ratio	Wins/games	Avg units	Avg bases	Avg score
NUS	64%	(64/100)	22.95	3.13	28.28
WarsawB	48%	(48/100)	18.32	1.98	15.31
UBC	57%	(57/100)	30.48	1.71	29.90
Uofa.06	88%	(88/100)	29.69	4.00	40.49
Average	64.25%	(64.25/100)	25.36	2.71	28.50

Table 7.4: *Experiment results from defensive field*

$$p_{defensive}(d) = \begin{cases} w_2 \cdot (-800 + 6.4 \cdot d) & \text{if } d <= 125 \\ 0 & \text{if } d > 125 \end{cases} \tag{7.11}$$

The use of a defensive field is a great performance improvement of the bot, and it now wins over 64% of the games against the four opponent teams (Table 7.4).

7.5.7 Local optima

To get stuck in local optima is a problem that is well-known and that has to be dealt with when using PF. Local optima are positions in the potential field that have higher potential than all their neighbouring positions. An agent positioned in a local optimum may therefore get stuck even if the position is not the final destination for the agent. In the first version of our bot, agents that had been idle for some time moved in random directions for some frames. This is not a very reliable solution to the problem since there are no guarantees that the agents will move out of, or will not directly return to, the local optima.

Thurau et al. (Thurau et al., 2004a) describes a solution to the local optima problem called *avoid-past potential field forces*. In this solution each agent generates a trail of negative potentials on previous visited positions, similar to a pheromone trail used by ants. The trail pushes the agent forward if it reaches a local optimum. We have introduced a trail that adds a negative potential to the last 20 positions of each agent. Note that an agent is not affected by the trails of other own agents. The negative potential used for the trail is set to -0.5.

The use of pheromone trails further boosts the result and our bot now wins 76.5% of the games (see Table 7.5).

7.5.8 Using maximum potentials

In the original bot all potential fields generated by opponent units were weighted and summed to form the total potential field which is used for navigation by our agents. The effect of summing the potential fields generated by opponent units is that the highest

Team	Wins ratio	Wins/games	Avg units	Avg bases	Avg score
NUS	73%	(73/100)	23.12	3.26	32.06
WarsawB	71%	(71/100)	23.81	2.11	27.91
UBC	69%	(69/100)	30.71	1.72	31.59
Uofa.06	93%	(93/100)	30.81	4.13	46.97
Average	76.5%	(76.5/100)	27.11	2.81	34.63

Table 7.5: *Experiment results from avoid-past potential field forces*

Team	Win %	Wins/games	Avg units	Avg bases	Avg score
NUS	100%	(100/100)	28.05	3.62	46.14
WarsawB	99%	(99/100)	31.82	3.21	47.59
UBC	98%	(98/100)	33.19	2.84	46.46
Uofa.06	100%	(100/100)	33.19	4.22	54.26
Average	99.25%	(99.25/100)	31.56	3.47	48.61

Table 7.6: *Experiment results from using maximum potential, instead of summing the potentials.*

potentials are generated from the centres of the opponent unit clusters. This make our agents attack the centres of the enemy forces instead of keeping the MSD to the *closest* enemy. The proposed solution to this issue is that, instead of summing the potentials generated by opponent units and bases, we add the highest potential any opponent unit or base generates. The effect of this is that our agents engage the closest enemy unit at maximum shooting distance instead of trying to keep the MSD to the centre of the opponent unit cluster. The results from the experiments are presented in Table 7.6.

7.5.9 A final note on the performance

Our results were further validated in the 2008 ORTS tournament, where our PF based bots won the three competitions that we participated in (Collaborative Pathfinding, Tankbattle, and Complete RTS). In the Tankbattle competition, we won all 100 games against NUS, the winner of last year, and only lost four of 100 games to Lidia (see Table 7.7 (Buro, 2008)).

Team	Total win %	Blekinge	Lidia	NUS
Blekinge	98	—	96	100
Lidia	43	4	—	82
NUS	9	0	18	—

Table 7.7: *Results from the ORTS Tankbattle 2008 competition.*

7.6 Fog of war

To deal with FoW the bot needs to solve the following issues; remember locations of enemy bases, explore unknown terrain to find enemy bases and units and handle dynamic terrain due to exploration. We must also take in consideration the increase in computational resources needed when designing solutions to these issues. To enable FoW for only one client, we made a minor change in the ORTS server. We added an extra condition to an IF statement that always enabled fog of war for client 0. Due to this, our client is always client 0 in the experiments (of course, it does not matter from the game point of view if the bots play as client 0 or client 1). Below follows the changes we made to deal with these issues.

7.6.1 Remember locations of the Enemies

In ORTS a data structure with the current game world state is sent each frame from the server to the connected clients. If fog of war is enabled, the location of an enemy base is only included in the data structure if an own unit is within visibility range of the base. It means that an enemy base that has been spotted by an own unit and that unit is destroyed, the location of the base is no longer sent in the data structure. Therefore our bot has a dedicated global map agent to which all detected objects are reported. This agent always remembers the location of previously spotted enemy bases until a base is destroyed, as well as distributes the positions of detected enemy tanks to all the own units.

The global map agent also takes care of the map sharing concerning the opponent tank units. However, it only shares momentary information about opponent tanks that are within the detection range of at least one own unit. If all units that see a certain opponent tank are destroyed, the position of that tank is no longer distributed by the global map agent and that opponent disappears from our map.

7.6.2 Dynamic Knowledge about the Terrain

If the game world is completely known, the knowledge about the terrain is static throughout the game. In the original bot, we created a static potential field for the terrain at the beginning of each new game. With fog of war, the terrain is partly unknown and must be explored. Therefore our bot must be able to update its knowledge about the terrain.

Once the distance to the closest impassable terrain has been found, the potential is calculated as:

$$p_{terrain}(d) = \begin{cases} -10000 & \text{if } d <= 1 \\ -5/(d/8)^2 & \text{if } d \in]1, 50] \\ 0 & \text{if } d > 50 \end{cases} \tag{7.12}$$

7.6.3 Exploration

Since the game world is partially unknown, our units have to explore the unknown terrain to locate the hidden enemy bases. The solution we propose is to assign an attractive field to each unexplored game tile. This works well in theory as well as in practice if we are being careful about the computation resources spent on it.

The potential $p_{unknown}$ generated in a point (x, y) is calculated as follows:

I. Divide the terrain tile map into blocks of 4x4 terrain tiles.

II. For each block, check every terrain tile in the block. If the terrain is unknown in ten or more of the (at most 16) checked tiles the whole block is considered unknown.

III. For each block that needs to be explored, calculate the Manhattan Distance md from the center of the own unit to the center of the block.

IV. Calculate the potential $p_{unknown}$ each block generates using Equation 7.13 below.

V. The total potential in (x, y) is the sum of the potentials each block generates in (x, y).

$$p_{unknown}(md) = \begin{cases} (0.25 - \frac{md}{8000}) & \text{if } md <= 2000 \\ 0 & \text{if } md > 2000 \end{cases} \tag{7.13}$$

7.6.4 Experiments, FoW-bot

In this experiment set we have used the same setup as in the Tankbattle except that now our bot has FoW enabled, i.e. it does not get information about objects, terrain, etc. that is further away than 160 points from all of our units. At the same time, the opponents has complete visibility of the game world. The results of the experiments are presented in Table 7.8. They show that our bot still wins 98.5% of the games against the opponents, which is just a minor decrease compared to having complete visibility.

It is also important to take in consideration the changes in the needs for computational resources when FoW is enabled, since we need to deal with dynamic terrain and exploration field. To show this we have run 100 games without FoW against team NUS and 100 games with FoW enabled. The same seeds are used for both. For each game we measured the average time in milliseconds that the bots used in each game frame and the number of own units left. Figure 7.6 shows the average frame time for both bots in relation to number of own units left. It shows that the FoW enabled bot used *less* CPU resources in the beginning of a game, which is probably due to that some opponent units and bases are hidden in unexplored areas and less potential fields based on opponent

Team	Wins ratio	Wins/games	Avg units	Avg bases	Avg score
NUS	100%	(100/100)	29.74	3.62	46.94
WarsawB	98%	(98/100)	32.35	3.19	46.70
UBC	96%	(96/100)	33.82	3.03	47.67
Uofa.06	100%	(100/100)	34.81	4.27	54.90
Average	98.5%	(98.5/100)	32.68	3.53	49.05

Table 7.8: *Experiment results when FoW is enabled for our bot.*

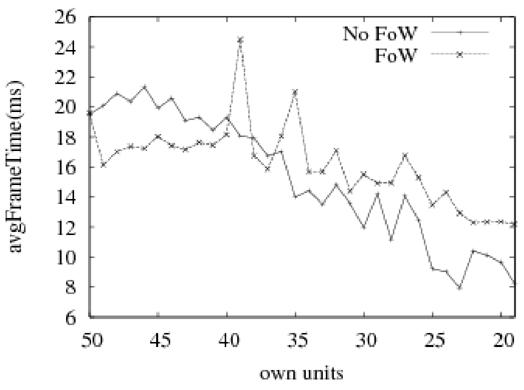


Figure 7.6: *The average frame time used for bots with perfect and imperfect information about the game world.*

units have to be generated. Later in the game the FoW-bot requires more CPU resources probably due to the exploration and the dynamic terrain fields.

In the next set of experiments we show the performance of the exploration field. We ran 20 different games in this experiment, each where the opponent faced both a bot with the field of exploration enabled, and one where this field was disabled (the rest of the parameters, seeds, etc. were kept identical). Figure 7.7 shows the performance of the exploration field. It shows how much area that has been explored given the time of the game. The standard deviation increases with the time since only a few of the games last longer than three minutes.

In Table 7.9, we see that the use of the field of exploration (as implemented here) does not improve the results dramatically. However, the differences are not statistically significant.

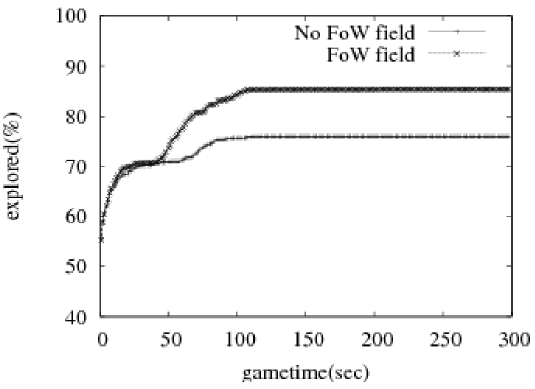


Figure 7.7: *The average explored area given the current game time for a bot using the field of exploration, compared to one that does not.*

Version	Won	Lost	Avg. Units	Avg. Bases
With FoE	20	0	28.65	3.7
Without FoE	19	1	27.40	3.8

Table 7.9: *Performance of the bot with and without field of exploration in 20 matches against NUS.*

7.7 Discussion

We have shown that the bot can easily be modified to handle changes in the environment, in this case both a number of details concerning the agents, the granularity, the fields, but also FoW. The results show that FoW initially decreases the need for processing power and in the end, it had a very small impact on the performance of the bot in the matches. However, this has to be investigated further. In Figure 7.7 we see that using the field of exploration in general gives a higher degree of explored area in the game, but the fact that the average area is not monotonically increasing as the games go on may seem harder to explain. One plausible explanation is that the games where our units do not get stuck in the terrain will be won faster as well as having more units available to explore the surroundings. When these games end, they do not contribute to the average and the average difference in explored areas will decrease. Does the field of exploration contribute to the performance? Is it at all important to be able to explore the map? Our results (see Table 7.9) indicate that it in this case may not be that important. However, the question is complex. Our experiments were carried out with an opponent bot that had perfect information and thus was able to find our units. The results may have been different if also the opponent lacked perfect information.

It is our belief that MAPF based bots in RTS games has great potential even though the scenarios used in the experiments are, from an AI perspective, quite simple RTS scenarios. In most modern commercial RTS games the AI (and human player) has to deal with base building, economics, technological development and resource gathering. However, we can not think of any better testbed for new and innovative RTS games AI research, than to test it in competitions like ORTS.

7.8 Conclusions and Future Work

In section 7.4 we introduced a methodology for creating MAPF based bots in an RTS environment. We showed how to deal with a gathering resources scenario in a MAPF based bot. Our bot won this game in the 2008 years' ORTS competition, but would have ended up somewhere in the middle in 2007 years' tournament. The bot had some problems with crashes and more work can be done here to further boost the result.

This was followed by Section 7.5 where we showed how to design a MAPF based for playing a tankbattle game. The performance of the first version of our bot was tested in the 2007 years' ORTS competition organized by the University of Alberta. The results, although not very impressive, showed the use of MAPF based bots had potential. A number of weaknesses of the first version were identified, solutions to these issues were proposed and new experiments showed that the bot won over 99% of the games against four of the top teams from the tournament. This version of the bot won the 2008 years' tournament with an almost perfect score of 98% wins.

Some initial work has been done in this direction. Our bot quite easily won the full

RTS scenario in the 2008 years' ORTS tournament, but more has to be done here. The full RTS scenario in ORTS, even though handling most parts of a modern RTS game, is still quite simple. We will develop this in the future to handle a larger variety of RTS game scenarios.

Another potential idea is to use the fact that our solution, in many ways, is highly configurable even in runtime. By adjusting weights of fields, the speed of the units, etc. in real time, the performance can be more or less changed as the game goes on. This can be used to tune the performance to the level of the opponent to create games that are more enjoyable to play. One of our next projects will focus on this aspect of MAPF based bots for RTS games.

CHAPTER

EIGHT

PAPER VII

An expandable multi-agent based architecture for StarCraft bots.

Johan Hagelbäck
Submitted for publication

8.1 Introduction

Real-Time Strategy (RTS) games provide many challenges both for human- and computer controlled (bot) players. The player usually starts with a command center and a number of workers. The workers must be used to gather one or more types of resources from resource spots and drop them off at the command center. The resources can in turn be used to construct buildings that expand your bases, or to build units that may attack the opponent or defend own base(s). It is also common that RTS games have technology trees where the player can invest resources in upgrades for units and/or buildings. The game usually ends when a player has destroyed all the buildings of the opponents which requires a lot of skill and micro-management by the player.

The gameplay can be divided into a number of sub problems:

Resource Gathering. Resources of one or more types must be gathered by the player to construct buildings, research upgrades and create units. This usually means that a number of workers have to move from the command center to a resource spot, gather resources, then return to the command center to drop them off. If workers can collide with each other the bot has to handle dynamic pathfinding and movement where a lot of workers move back and forth at an often relatively small area. In some games, for example StarCraft, workers are transparent in the sense that they can move through each other when gathering resources. This can reduce the computational resources needed for dynamic pathfinding.

Constructing buildings and units. Each building costs a fixed amount of resources to create. There are usually a number of buildings a player have to construct, for example barracks and factories to be able to produce combat units. There are also usually a number of optional buildings that give access to special units and/or upgrades that are beneficial. The player has to decide which buildings to construct and in what order, and which units to build for defense and attack.

Base planning. This involves how to place buildings in a base. Which buildings should be placed in the middle of the base for extra protection? Where should defensive turrets be placed? How close to each other shall buildings be placed to avoid getting a too spread out or too crowded base?

Upgrading. The player usually has a wide range of upgrades and research to choose from, all of which take time and cost resources. Some upgrades make units of a specific type stronger, while others can unlock special abilities for units or allow construction of new unit types. The player has to chose which upgrades to execute and in which order. In general the player should only perform the upgrades that are beneficial for his/her playstyle. It is a waste of resources to upgrade units that the player rarely or never uses.

High-level combat tactics. To win in an RTS game it is often a good strategy to create a diverse army consisting of several unit types. In a well-balanced game there are no or very few (but expensive) units that are very strong against all opponent units, instead the different unit types are designed in a rock-paper-scissor like fashion where a unit can be very good against one type of units but quite weak against others. Table 8.1 shows counters (which units are strong against each unit type) for all Protoss units in StarCraft (RPGClassics, 2011). The extremely powerful units are usually only available late in the game after numerous upgrades, and are often very expensive to create. The high-level combat tasks involves:

- Setup of defensive and offensive squads.
- From where an attack at the enemy should be launched.
- Finding good spots to defend own bases.
- When to launch an attack at the enemy.

Micro-management. In a combat situation each individual unit should strive to maximize its utility by targeting opponents it is effective against, make use of special abilities when needed and defend weak but strong-hitting ally units.

Protoss Unit	Protoss Counter
Probe	Zealot
Zealot	Carrier
Dragoon	Zealot
High Templar	Zealot or Dark Templar
Dark Templar	Observer + Zealot
Shuttle	Scout
Reaver	Carrier
Observer	Observer + Scout or Observer + Corsair
Scout	Dragoon
Corsair	Dragoon
Carrier	Scout
Arbiter	Observer + Corsair
Archon	Zealot
Dark Archon	Zealot

Table 8.1: *Counters against Protoss units in StarCraft.*

An example from StarCraft is the Protoss unit Archon. It is a very powerful unit with 10 health points but 350 shield points, an effective health of 360 points which is more than most units in the game. The Terran unit Science Vessel has an ability called EMP Shockwave. This ability destroys the force shield around a Protoss unit, reducing the effective health of the Archon to 10 in a single strike.

To maximize the potential of each unit requires a lot of micro management by the player. Human players are usually assisted by the AI in that units have some logic to decide which opponent units to target. Units must also know how to navigate in the game world and avoid colliding with other units, obstacles and terrain. Many units also have special abilities that can be used to increase the strength of the unit for a short duration or help counter a specific threat.

We propose an expandable and modifiable multi-agent based bot for StarCraft which is tested in a number of matches against bots participating in the AIIDE 2010 StarCraft bot tournament. The bot is released as open source and is available for download[1]. To interact with the StarCraft game engine the bot uses BWAPI (BWAPI, 2009)

The rest of the paper is organized as follows; first we go through some related work. It is followed by a description of the bot and its architecture. We continue with describing some experiments to evaluate the performance of the bot, and then discuss how the proposed architecture can meet the design goals of being flexible and expandable. At

[1]BTHAI Project - http://code.google.com/p/bthai/

last we give some pointers to future work.

8.2 Related Work

Game worlds in RTS games are highly dynamic (e.g. due to the large number of moving objects) which make tasks such as navigation somewhat difficult using conventional pathfinding methods. Still, pathfinding with the famous A* algorithm is probably the most common technique used for navigation.

Extensive work has been done to modify A* to work better in highly dynamic worlds. Silver proposes an addition of an extra time dimension to the pathfinding graph to allow units to reserve a node at a certain time (Silver, 2006). The work of Olsson addresses the issue of changes in the pathfinding graph due to the construction or destruction of buildings (Olsson, 2008). Koenig and Likachev have made contributions to the field with their work on Real-Time A* (Koenig, 2004; Koenig & Likhachev, 2006).

Potential Fields is a concept originating from robotics. It was first introduced by Khatib for real-time obstacle avoidance for manipulators and mobile robots (Khatib, 1986). It works by placing attracting or repelling charges at important locations in the virtual world. An attracting charge is placed at the position to be reached, and repelling charges are placed at the positions of obstacles. Each charge generates a field of a specific size. A repelling field around obstacles are typically small while the attracting field of positions to be reached has to cover most of the virtual world. The different fields are weighted and summed together to form an aggregated field. The total field can be used for navigation by letting the robot move to the most attracting position in its near surroundings. Many studies concerning potential fields are related to spatial navigation and obstacle avoidance, for example the work by Borenstein and Massari (Borenstein & Koren, 1991; Massari et al., 2004). Alexander describes the use of fields for obstacle avoidance in the games Blood Wake and NHL Rivals (Alexander, 2006). Johnson describes obstacle avoidance using fields in the game The Thing (Johnson, 2006).

In previous papers we have extended the use of *Potential Fields* to not only handle local obstacle avoidance, but to replace the pathfinding algorithm and be used for all navigation tasks (see Paper I, Paper III, Paper V and Paper VI). Figure 8.1 shows an example of how Potential Fields (PFs) can be used for navigation in a game world. A unit (bottom left corner) moves to its destination at E. The destination has an attractive charge (light areas) that gradually fades to zero (dark areas). Mountains (black areas) and two obstacles (white circles) generate small repelling fields (darker areas) for obstacle avoidance.

We have also shown that a *Multi-Agent Potential Field* based solution can match and even surpass the performance of traditional pathfinding based methods in the open source RTS game ORTS (see Paper III, Paper V and Paper VI). There are however issues that are not fully dealt with, for example how to handle complex maps with lots of chokepoints and narrow passages.

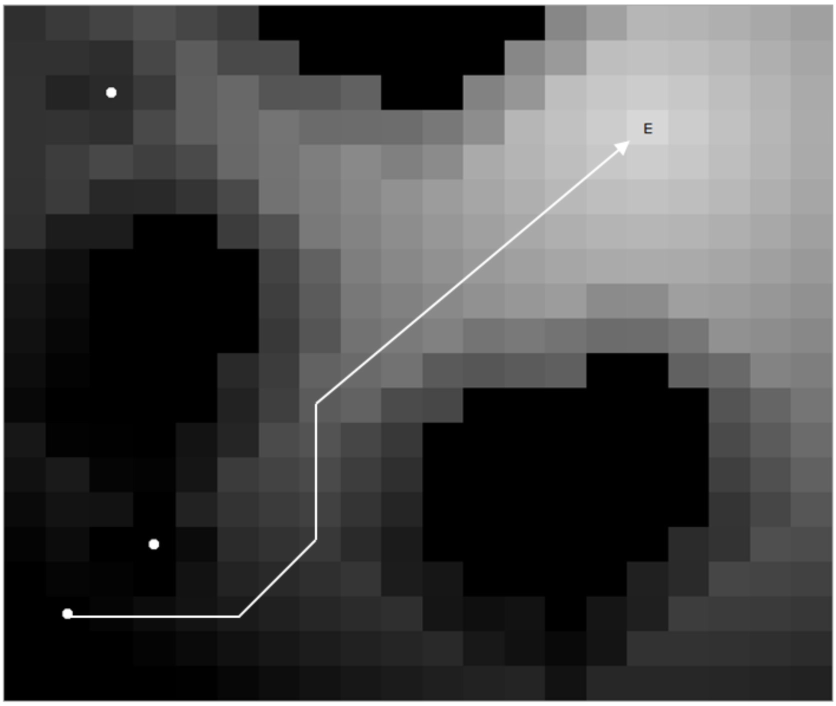


Figure 8.1: *Example of PFs in a game world.*

To make high-level strategical decisions is another challenge. Combat units have to be gathered in groups where each group has different goals, for example attacking the enemy, exploring the game world or defending own buildings and resource income. Each group can consist of different unit types with different strengths and weaknesses.

Strategical decisions are usually implemented using a Command Hierarchy architecture (Reynolds, 2002; Pittman, 2008). This means that the decisions are separated into several layers of abstraction. At the highest level we have a General or Commander agent. This agent is responsible for high-level decisions such as when and where to engage the enemy, and how to defend own bases. The General gives order to a Captain agent. The Captain agent is responsible for executing the General's orders. This is done by commanding a number of squads (a squad is a group of combat units). An example can be to send two heavy squads to attack an enemy base from the front, and send a third squad to attack the enemy from the flank or rear. When a Captain agent orders a squad to do something, a Sergeant agent is responsible for executing the order. The Sergeant moves the units in his squad to attack or defend in a good way. An example is to keep artillery units in the back and defend them with soldiers. At the lowest level we have the Soldier agent. It is responsible for handling a single unit. It involves navigation, when to stop and fire at an enemy, and more.

Another important task is in which order to construct buildings. The most common technique is to use a buildorder (Schwab, 2009), which is a static list of what to build and in which order. An example is build two Barracks, one Refinery, after that an Academy and so on. The buildorder can be adapted at runtime. A simple example is Supply Depots in StarCraft. A single Supply Depot can handle a specific number of units, and when that space is filled a new Supply Depot must be created. This can be handled by just adding a Supply Depot first in the buildorder list well in time before the supply limit is reached.

Lately goal-oriented planners have gained some ground in handling strategic decisions (Cerpa, 2008; Dill, 2006; Pittman, 2008).

A third challenge is terrain analysis, a task that human players usually are good at but which can be tricky for a bot. A good general must understand his terrain to win battles. The same goes for RTS games; the bot must be able to identify important terrain features such as chokepoints, good places for ambush attacks etc. A common technique is for level designers to help the game AI by placing hints in the map data, i.e. the map itself tells the AI where a good chokepoint is and the AI does not have to read and interpret the terrain, just use the hints placed by the designers (Schwab, 2009).

8.3 Bot architecture

The complexity of RTS games makes them interesting test beds for education and research in artificial intelligence. To create an RTS bot from scratch is a complex and very time consuming task. One goal of our bot is to be able to play StarCraft games reasonably well, but the main goal is to have an architecture that can be easily used and modified for future education and research within game AI.

To meet the main goal a number of requirements on the bot were defined:

- The bot should be able to play on a majority of the StarCraft maps. It will not support maps where there are no groundpath between the start locations.

- The bot should be able to play all three races available in the game.

- Low-level and high-level tactics should be separated so changes in one level have a very small chance of breaking the logic in other levels.

- Basic functions such as move and attack should work for all units without adding specific code for a unit type. It should also be possible to add special logic for a unit type to handle special abilities.

- Units should be grouped in squads. Different squads may have different goals and responsibilities.
- The bot should be able to have different build orders and squads setup for each player/opponent combination, for example Terran vs. Zerg. Changing build orders and squads setup shall not require any changes to the logic.

The architecture of the bot is shown in Figure 8.2. It is divided into three modules; *Managers*, *CombatManagers* and *Agents*. Each module is described in detail below.

8.3.1 Agents

Each unit and building in the game is handled by an agent. We chose to use an inheritance system in three levels to be able to have common logic for all combat units and buildings at one level, and unit type specific code at a lower level.

BaseAgent is the abstract base class for all agents. It contains logic useful for both units and buildings, for example to check if the unit is under attack or has been damaged.

StructureAgent is the base agent for all buildings. It contains logic for producing units, do research and build upgrades. Most buildings are instantiated using *StructureAgent*. Exceptions are special buildings like for example the Terran Bunker that some ground units can enter for extra protection. *StructureAgent* uses the *ResourceManager* agent to check if there are enough resources available for an upgrade or research, and the *Commander* agent to check if a unit of a certain type shall be trained or not.

UnitAgent is the base agent for all mobile units except workers. A unit instantiated from *UnitAgent* can do all basic tasks such as move, attack and defend but no special abilities can be used. Unit types with special abilites have their own agent implementation that extends from *UnitAgent*. In Figure 8.2 MarineAgent, SiegeTankAgent and WraithAgent are shown but in the implementation almost all unit types have an own implementation.

WorkerAgent handles all types of workers; Terran SCVs, Protoss Probes and Zerg Drones. It contains logic for gathering resources, finding spots to create new buildings at, move to a spot and construct a new building, and in the case of Terran repairing damaged buildings and tanks. The agent is implemented using a finite-state machine.

8.3.2 Managers

A manager is a global agent that handles some higher level tasks. The system has one instance of each manager, and that instance can be accessed from all agents and classes within the system. Code wise this has been solved by implementing each manager agent as a Singleton class.

BuildPlanner contains a build order list of what buildings to construct and in which order. Each update frame the *BuildPlanner* asks the *ResourceManager* if there are

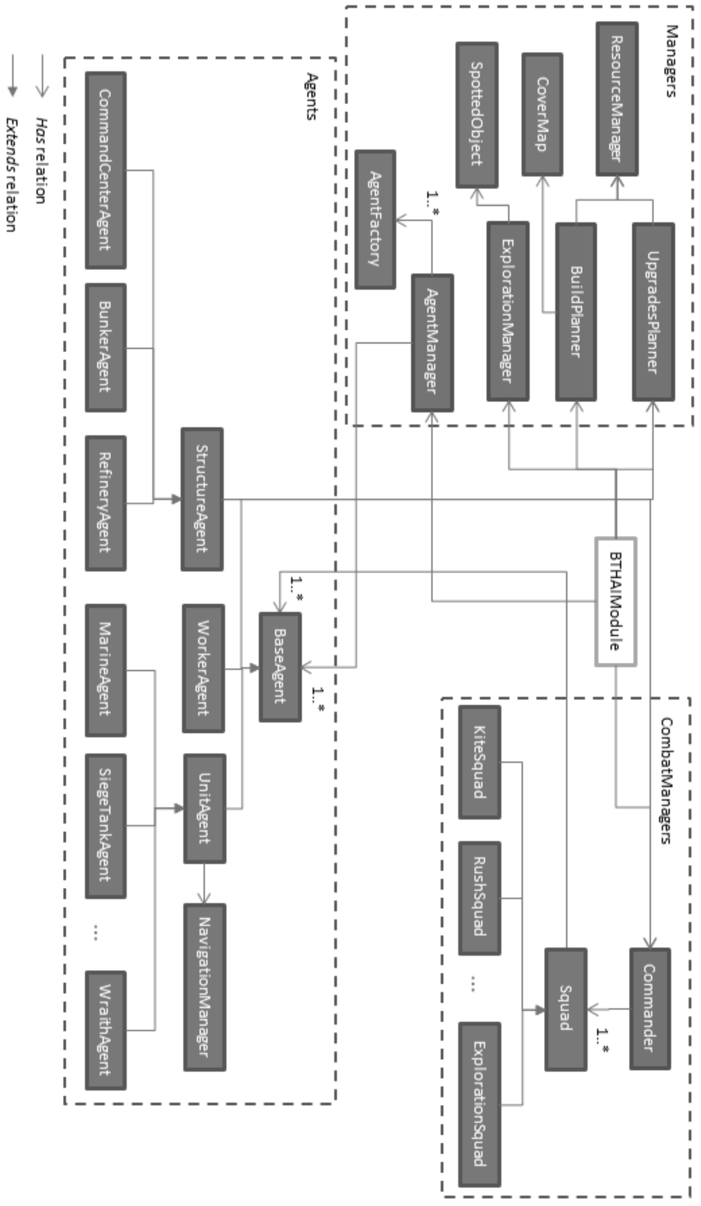


Figure 8.2: *The bot architecture.*

enough resources available to construct the first building in the list. If there are, a free worker is assigned to create the building. A free worker means a worker that is either idle (should rarely happen) or a worker gathering minerals. Once assigned the resource cost of the building is locked in the *ResourceManager* to avoid other tasks using up the required resources. The resource costs in the game are paid when the construction of the building is started and not when a build order is issued. Once the construction is started the resource lock is released, the building is removed from the build order list and is added as an agent. If a building agent is destroyed, the type is added as a new entry first in the buildorder list.

When finding a suitable spot for a new building the *CoverMap* agent is used. This agent contains a grid map of the game world where each tile is either blocked or free. Buildings cannot be placed on blocked tiles. A tile can be blocked if it has a terrain type that does not allow buildings to be placed upon it, or if a building has already been created on the tile. When searching for a buildspot the algorithm starts on a specific start location and searches outwards for the free buildspot closest (using Euclidean distance) to the start location. The start location is where the last Command Center was built, with the exception of defensive buildings such as Terran Bunkers or Protoss Photon Cannons. Instead they use the closest choke point as start location since that is where the enemy ground units will attack from. Some defensive structures such as Protoss Photon Cannons are effective against air units and could be valuable inside a base to deal with enemy flight rush attacks, but this is currently not handled by the bot. To find choke points we use the BWTA Broodwar Terrain Analyzer module in BWAPI. If two build spots have the same distance to the start location, the first one found is used. Once a build spot has been found it is temporary blocked. When the building is completed the temporary block is changed to a full block. If the building for some reason cannot be completed for example due to lack of resources, the temporary block is released. If an own building is destroyed the positions it blocked are also released.

The build order lists are read from text files during the startup of the bot. The bot uses a general buildorder for each playing race (Terran, Protoss and Zerg). It is also possible to add specific build orders for each player/opponent combination, for example Terran vs. Protoss. The text files contain ordered lists of the buildings to be constructed, using the BWAPI name of the buildingtypes. Figure 8.3 shows the BNF grammar for the buildorder text file, and the configuration used in the experiments are shown in Table 8.4, Table 8.9 and Table 8.12.

To further explain the flow of the system Figure 8.4 shows a use case diagram of how agents interact when a new building is constructed. Figure 8.5 shows a use case diagram of creating a new unit.

UpgradesPlanner handles lists of upgrades/techs similar to a build order list. A building in StarCraft can have several upgrades and/or techs available. For each of them, the *StructureAgent* handling the building asks the *UpgradesPlanner* if the upgrade/tech shall be executed. If a yes is received the upgrade or tech is executed and removed from

```
[buildorder-list] ::= [entry]
[buildorder-list] ::= [entry] [buildorder-list]
[entry] ::= [type] [EOL]
[type] ::= Terran_Academy | Terran_Armory |
        ... | Terran_Supply_Depot |
        Protoss_Arbiter_Tribunal |
        Protoss_Assimilator | ... |
        Protoss_Templar_Archives |
        Zerg_Creep_Colony | Zerg_Defiler_Mound
        | ... | Zerg_Ultralisk_Cavern
```

Figure 8.3: *BNF grammar for the buildorder text file version 2.3.*

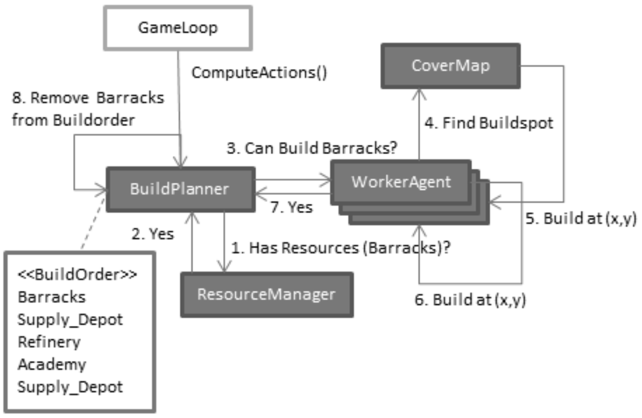


Figure 8.4: *Use case: Create a new building.*

the upgrades/techs list. Before accepting an upgrade/research tech order the *Upgrades-Planner* asks the *ResourceManager* if there are enough resources available.

The upgrades/techs list uses a priority system to distinct between important and less important upgrades and techs. An upgrade/tech of priority 1 is most important, 3 least important and 2 in between. An upgrade/tech in level 2 is never executed until all upgrades/research in level 1 has been completed. Similarly upgrades/techs in level 3 are never completed until all in level 1 and 2 has been completed. The order in the list is not important in contrast to how the build order list works. Whenever a tech building is idle, it checks all available upgrades it can do and asks the UpgradePlanner if any of them shall be executed. We believe the grouping in priority is more intuitive than using an ordered list, since it can be difficult to see when a specific upgrade or tech is available

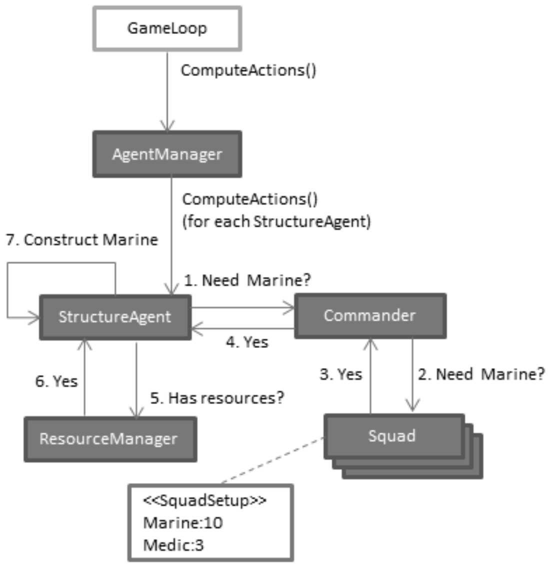


Figure 8.5: *Use case: Create a new unit.*

in the game and to avoid a badly placed upgrade/tech to lock the whole list.

The upgrades/techs list is read from text files at the startup of the bot. There is a general list for each race, but it is also possible to add specific lists for each player/opponent combination. The BNF grammar for the upgrades/techs text file is shown in Figure 8.6, and the configuration files used in the experiments are shown in Table 8.5, Table 8.10 and Table 8.13.

```
[upgrades-list] ::= [entry]
[upgrades-list] ::= [entry] [upgrades-list]
[entry] ::= [type]:[priority] [EOL]
[type] ::= U-238_Shells |  Tank_Siege_Mode
       Terran_Infantry_Weapons | ... | Stim_Packs
[priority] ::= 1 | 2 | 3
```

Figure 8.6: *BNF grammar for the upgrades/techs text file version 2.3.*

To make the system flow more clear Figure 8.7 shows a use case of how agents interact when executing an upgrade or researching a new tech.

AgentManager is a container class for all agents in the game. It contains a list of

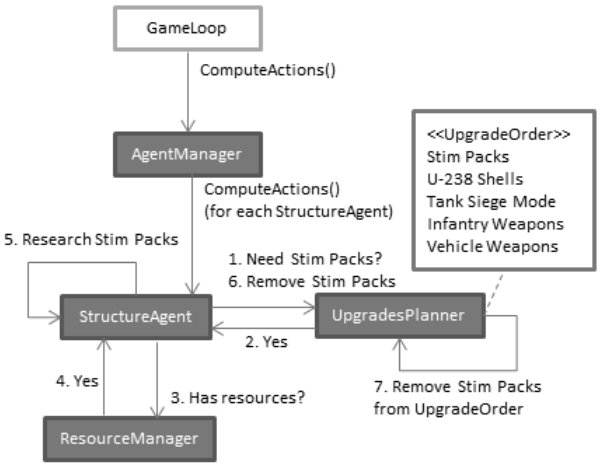


Figure 8.7: *Use case: Execute an upgrade.*

active agents, adds new agents when a unit or building has been created, and removes agents when a unit or building has been destroyed. The agent also contains a number of decision support methods, for example calculating the number of enemy units within a specific area or how many own units of a specific unit type that have been created. It is also responsible for calling an *Update* method on each agent at each update frame.

ExplorationManager is responsible for all tasks involved in exploring the game world. It controls a number of scout units, and decides where each unit should go next. Scouting can be performed by any unit type, but some are more appropriate than others. Flying units are best but are not available at the start of a game, and a fast moving ground unit is better than a slow one.

The exploration logic uses the regions from the BWTA (Broodwar Terrain Analyzer, included in BWAPI) module. When an explorer unit reaches the center of a region, it checks all adjacent regions and moves to the one that it has been the longest time since an own unit visited.

ResourceManager is responsible for managing the resources (minerals and gas) available to the player. Resources are gathered by the workers and are needed for new buildings, new units, upgrades and research. Each agent that can produce something that costs resources must ask the *ResourceManager* if there are enough resources available to complete the order.

Currently the only limitation in resource management is that once a unit production building is completed, build orders are not allowed to reduce the total amount of minerals below 150. This limitation is used to make sure creating new buildings does not use up all resources and slow down the production of units.

8.3.3 CombatManagers

For combat tasks the bot uses a command hierarchy in three levels (*Commander*, *Squad* and *UnitAgent*) as shown in Figure 8.2.

The *Commander* agent is responsible for decisions at the highest level. It is a global agent that can be accessed by all agents in the system (similar to non-combat managers). The tasks to be handled by the *Commander* are:

- Launch attacks against the enemy.
- Find a good spot to attack an enemy base at.
- Decide which squads, if any, shall assist buildings or units under attack.
- Decide good spots for defending own base(s).
- If playing Terran, assign workers to repair damaged buildings or important mechanical units.

The *Commander* is implemented as a rule based system. If certain conditions are met, something happens. This could be attacking an enemy base or something else. The rules are hard-coded and cannot be modified without recompiling the bot.

The *Commander* is in charge of one or more squads. Each squad is in turn responsible for a number of unit agents that control the StarCraft units attached to the squad. The *Squad* agent is a general agent for handling standard tasks such as move to a position, defend a position or attack a target. To create more specialized squads it is possible to add new squad implementations that extend the basic *Squad*. In Figure 8.2 KiteSquad, RushSquad and ExplorationSquad are shown but there are more special squads available in the system.

A squad can consist of a single unit up to almost any number and many combinations of units. The squad agent is responsible for coordinating its units to attack in an efficient manner. This involves moving the squad as a group and not let the units spread out too much, put long range units in the rear when attacking, defending weak but powerful units such as siege tanks, and more.

Squads setup are read from text file at the startup of the bot. Each playing race has a general text file, and it is also possible to add a specific text file for each player/opponent combination. Figure 8.8 shows the BNF grammar for the language used to define squads setup, and the configurations used in the experiments are shown in Table 8.6, Table 8.11 and Table 8.14.

Below we explain the non-self-explanatory elements in a squad setup entry in more detail:

```
[squad-setup] ::= <start> [EOL]
           Type=[squad-type] [EOL]
           Move=[move-type] [EOL]
           Name=[name] [EOL]
           Priority=[priority] [EOL]
           ActivePriority=[active-priority] [EOL]
           Offense=[offense-type] [EOL]
           <setup> [EOL]
           [unit-list] [EOL]
           <end> [EOL]

[squad-type] ::= Offensive | Defensive |
           Exploration | Support | Rush | Kite |
           ReaverBaseHarass | ChokeHarass |
           ShuttleReaver
[move-type] ::= Ground | Air
[name] ::= Lurker[string] | Devourer[string] |
           [string]
[priority] ::= 1 | 2 | 3 | ... | 1000
[active-priority] ::= 1 | 2 | 3 | ... | 1000
[offense-type] ::= Optional | Required

[unit-list] ::= [unit]
[unit-list] ::= [unit] [unit-list]
[unit] ::= Unit=[unit-type]:[amount] [EOL]
[unit-type] ::= Terran_Battlecruiser |
           Terran_Dropship | ... | Terran_Wraith
           | Protoss_Arbiter | Protoss_Archon
           | ... | Protoss_Zealot |
           Zerg_Defiler | Zerg_Devourer |
           ... | Zerg_Zergling
[amount] ::= 1 | 2 | 3 | ... | 200
```

Figure 8.8: *BNF grammar for the squads setup text file version 2.3.*

- *Type*. The basic squad types are *Offensive*, *Defensive*, *Exploration* and *Support* (non-attacking units such as Protoss Observer). In addition there are a number of special squads available. These are *Rush* (rushes the enemy base as soon as the squad is filled), *Kite* (a rush squad that uses a hit-and-run tactic), *ShuttleReaver* (Protoss Shuttle transporting slow moving Protoss Reavers), *ReaverBaseHarass* (Protoss Shuttle that drops Protoss Reavers in the enemy base) and *ChokeHarass* (guards chokepoints around an enemy base to slow down expansion).

- *Name*. An arbitrary name of the squad. If a squad containing Zerg Hydralisks has a name starting with *Lurker*, the hydralisks will be morphed to Zerg Lurkers. If a squad containing Zerg Mutalisk has a name starting with *Devourer*, the mutalisks will be morphed to Zerg Devourers. Otherwise Zerg Mutalisks will be morphed to Guardians.

- *Priority*. A priority value (integer between 1 and 1000) of the squad. Low values have high priority and high values low priority. A squad of priority *n+1* will never be filled until all squads of priority *n* or lower are filled.
- *ActivePriority*. Once a squad has been filled with units its *Priority* value is set to the *ActivePriority* value. A squad with *Priority* of 1000 will never be filled with units, i.e. it is considered inactive. *ActivePriority* can be used to create one time squads used for for example rush attacks.
- *OffenseType*. The bot will launch an attack on the enemy when all *Required* squads are filled with units.

Special abilities have been implemented for most units. Examples are Terran Marines who can enter bunkers and use Stim Packs when being attacked, Siege Tanks enter siege mode when there are ground targets within range and Science Vessels use EMP Shockwave on shielded Protoss units.

To show the flow of the system Figure 8.9 contains a use case diagram for how agents are communicating when ordering units to attack a target.

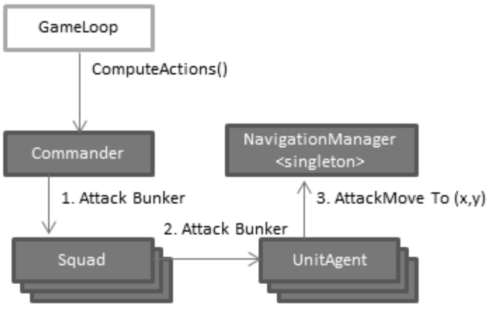


Figure 8.9: *Use case: Attack a target.*

8.3.4 Navigation and pathfinding

The navigation technique used in the bot combines both pathplanning and potential fields to get the best from both worlds.

One of the most interesting benefits of using potential field based navigation is its ability to produce interesting emergent behavior. By combining an attracting field with the highest potential at the maximum shooting distance around enemy units with a small repelling field around own units our forces surround the enemy at an appropriate range. This effect is possible to create by just using pathplanning techniques, but it is much more difficult to implement and generalize than when using potential fields. How potential fields are used in RTS games is described in more detail in Papers I, III and V.

Pathplanning techniques such as A* always find the shortest path between two positions. Agents navigating with potential fields can get stuck endlessly in local maxima. In previous work (see Papers I and III) we proposed the use of a pheromone trail to solve this issue. This will avoid units getting stuck endlessly, but cannot guarantee that it might take some time for units to get out of the local maxima. The problem gets worse the more complex a map is. In the work we used the ORTS game. Maps in ORTS tend to have very large open spaces and few areas with complex and narrow terrain. This is ideal for a potential field based navigation system. StarCraft maps are usually a bit more complex with narrow paths, chokepoints and islands which can cause problems for a navigation system based on potential fields only.

Our proposed method uses the built-in A* pathfinder in StarCraft when moving units between different positions, and once an own agent is within sight range of enemy units or buildings it switches to navigate using potential fields. When looking at the bot in action, units are moving on a line when travelling towards the enemy base and spread out to surround the enemy once own units get in sight range. Figure 8.10 shows a screenshot of navigation using potential fields where a number of Siege Tanks form an arc to surround the enemy.

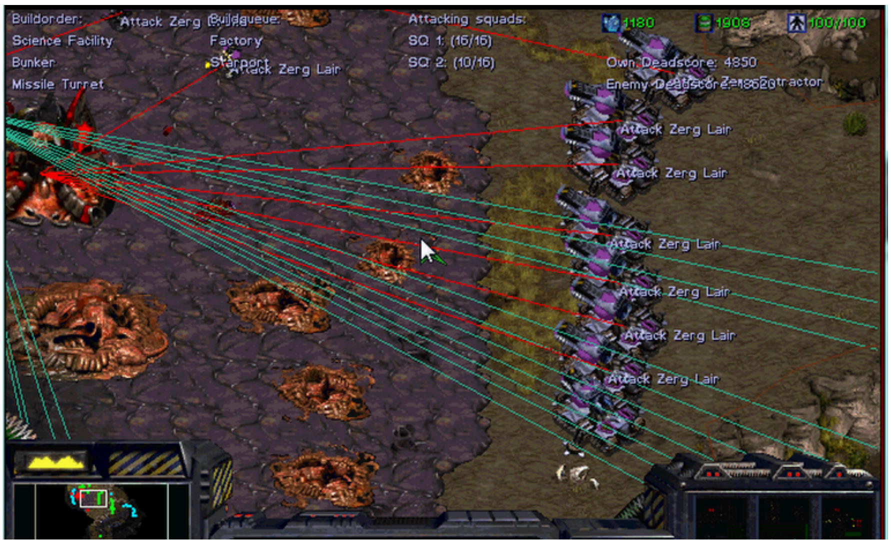


Figure 8.10: *Screenshot: Terran Siege Tanks surround the Zerg base.*

8.4 Experiments and Results

We have conducted experiments with the bots participating in Tournament 4 at the AI-IDE 2010 StarCraft bot competition (AIIDE 2010 StarCraft Competition, 2010). In total 17 bots participated including an early version of BTHAI. Out of the 16 opponents we were able to get 12 of them running on the latest version of BWAPI that BTHAI now is using. For the experiments we have chosen the following three maps:

- *Python*. This is a 4-player map that was part of the map pool in the tournament.
- *Destination 1.1*. This is a 2-player map that was part of the map pool in the tournament.
- *Benzene*. This is a 2-player map that was not in the tournament, but was in the pool for 2011 years' tournament. The reason to choose this map was to see if any bot performed significantly worse on this map which could indicate that the bot is coded to work specifically on the tournament maps.

For each opponent/map combination two games were played. All bots were facing BTHAI Protoss, BTHAI Terran and BTHAI Zerg. 18 games were played against each opponent bot, which would be in total 216 games. We were able to complete 198 games since not all opponent bots worked on all three maps. In all experiments version 2.3b of BTHAI was used.

The following tactics are used by BTHAI:

- BTHAI Terran tries to win with brute force. Large squads with a mix of Siege Tanks, Marines, Medics and Goliaths are created before launching an attack against the enemy. Later in the game the ground squads are supported by Battle Cruisers. BTHAI Terran has the slowest expansion rate of the three races. The effectiveness of the attacking squads relies heavily on sieged Siege Tanks. See Table 8.4, Table 8.5 and Table 8.6 for configurations.
- BTHAI Protoss tries a rush against the enemy with cloaked (invisible by most enemy units) Dark Templar units to hopefully slow down the expansion pace of the enemy. The attacking force consists of squads with Zealots and Dragoons supported by Observers to spot enemy cloaked units. The bot also tries to drop Reavers inside the enemy base to destroy vital buildings. BTHAI Protoss has a slightly higher expansion rate than BTHAI Terran. See Table 8.9, Table 8.10 and Table 8.11 for configurations.
- BTHAI Zerg rushes the enemy with small squads of Lurkers that cloak upon seeing enemy units. Lurkers can be constructed quite early in the game and especially against Terran and Protoss they can attack the enemy base before detectors (which can spot cloaked units) are built. If the opponent has not been beaten by the Lurker

rush a large air-heavy force is created before launching an attack against the enemy. BTHAI Zerg relies heavily on Lurkers and its defenses is pretty weak. It has the highest expansion rate of the three races. See Table 8.12, Table 8.13 and Table 8.14 for configurations.

The results are presented in Table 8.2. This is followed by a brief description of the opponent bots.

Opponent	Race	Wins	Opponent	Race	Wins
BroodwarBotQ	Protoss	0/6	Massexpand	Protoss	3/6
	Terran	1/6		Terran	0/6
	Zerg	5/6		Zerg	0/6
Chronos	Protoss	0/6	Omega	Protoss	0/6
	Terran	0/6		Terran	0/6
	Zerg	2/6		Zerg	5/6
HIJ	Protoss	6/6	Overmind	Protoss	0/4
	Terran	5/6		Terran	0/4
	Zerg	6/6		Zerg	0/4
JaReD	Protoss	2/2	Skynet	Protoss	1/6
	Terran	2/2		Terran	0/6
	Zerg	2/2		Zerg	0/6
Krasi0	Protoss	0/6	UAlberta	Protoss	0/6
	Terran	0/6		Terran	0/6
	Zerg	0/6		Zerg	0/6
Manolobot	Protoss	6/6	ZotBot	Protoss	0/6
	Terran	6/6		Terran	0/6
	Zerg	6/6		Zerg	6/6

Table 8.2: *Experiment results*

BroodwarBotQ, playing Protoss, gathers a large force of Dragoons in the middle of the map while expanding aggressively. The Dragoon force is usually able to reach any of the bases that are under attack quickly enough to save it from being destroyed. The bot was able to beat BTHAI playing Protoss and Terran quite easily. Note that many games had to be replayed due to BroodwarBotQ crashing. BTHAI Zerg was very effective against BroodwarBotQ, which had problems spotting the cloaked Lurkers.

Chronos, playing Terran, uses a very effective hit-and-run tactics with cloaked flying Terran Wraiths hitting targets in the outskirts of the enemy base. This makes it very difficult to expand the base and to keep workers gathering resources alive. Chronos also uses Vultures that drop Spider Mines all around the game world. BTHAI was not able to win any game against Chronos as Protoss or Terran, but won 2 of 6 games playing Zerg.

The success of Zerg was mainly due to Chronos having trouble spotting the cloaked Lurkers sneaking into the enemy base.

HIJ, playing Terran, uses a flying Engineering Bay building as a shield when attacking the enemy. The idea is that the enemy will waste firepower on attacking the building flying in front, while the Siege Tanks further back causes much damage to the enemy. BTHAI did not have any problems winning against HIJ and only lost one out of 18 played games.

JaReD, playing Terran, quickly falls behind in resource gathering and construction and was no match for BTHAI. We were only able to get JaReD to work on the Destination map. The other maps cause error messages and the bot does not construct any buildings or units.

Krasi0, playing Terran, expands very fast and has an effective defense. When an almost total domination of base locations have been taken the enemy is attacked. Krasi0 ended up on second place in the AIIDE 2010 tournament, beaten in the final by Overmind. Krasi0 won against BTHAI in all games.

Manolobot, playing Terran, is trying to keep pressure on the opponent by constantly attacking with small forces. This is easily handled by BTHAI which won all of the 18 games.

Massexpand, playing Zerg, tries an early rush with Zerlings while quickly trying to expand the base. As soon as possible flying Guardians are sent to harass the enemy base by attacking units and buildings in the outskirts. BTHAI playing Terran and Zerg had trouble defending against the early rush, while BTHAI Protoss was more successful with 3 wins on 6 games. Note that Massexpand have a habit of crashing on longer games so several games had to be replayed. In these games BTHAI usually was able to repel the initial Zergling rush and had a better chance of winning, so the results are slightly misguiding.

Omega, playing Protoss, expands effectively and tries to win by outnumbering the enemy. Omega won easily against BTHAI playing Protoss and Terran, but had too much trouble defending against the Lurker rush used by BTHAI Zerg.

Overmind, playing Zerg, rushed the enemy base very early with Zerglings and a few workers. The workers steal the opponent gas vein by building an Extractor in the opponent base, while the Zerglings runs around trying to kill every enemy worker. This proved to be an effective strategy and Overmind ended up the winner in the AIIDE 2010 tournament. BTHAI lost against Overmind in all games. Note that Overmind was not able to play the Benzene map due to not being able to handle non-ASCII characters in the map name.

Skynet, playing Protoss, is very effective at whatever it is doing. In the AIIDE 2010 tournament it was unlucky to be drawn against Overmind and Krasi0, two of the best bots in the tournament. BTHAI did not stand a chance against Skynet with only one win in 18 games.

UAlberta, playing Zerg, keeps a constant pressure on the enemy by sending group after group to harass the enemy base. BTHAI lost all games against UAlberta.

ZotBot, playing Protoss, builds up a strong defense and tries to harass the enemy by sending cloaked Dark Templars to their base. A full-scale attack is performed when a very strong force has been built. BTHAI playing Protoss or Terran have problems defending against the cloaked Templars, while BTHAI playing Zerg easily wins over ZotBot.

The total results are presented in Table 8.3. BTHAI Zerg was most successful with a win ratio of 48%, and the bot had an average of 33% wins.

Race	Wins	Loss	Win ratio
Protoss	19	66	29%
Terran	14	66	21%
Zerg	32	66	48%
Total	65	198	33%

Table 8.3: *Total results for BTHAI.*

8.5 Discussion

The experiments show us that BTHAI in its current form cannot compete with the best bots available, it is rather somewhere in the middle with at an average 33% wins. Most effective was the bot when playing Zerg with 48% wins (see Table 8.3). Our main goal of the project was however not to win as many games as possible, but rather to create a bot with an architecture that allows for modification at several levels of abstraction without requiring major code changes.

The use of text files for buildorders, upgrades, techs and squads setup makes it easy to completely change the strategy without needing to recompile the bot. This also makes it easier for non-coders to make additions to the bot. Since BTHAI can use separate text files for each player/opponent combination it is for example possible for Terran to use different strategies against Protoss, Terran and Zerg opponents. To prove this point we made a second experiment against the Massexpand bot. This time we modified the buildorder and squad text files to counter the early rush strategy used by Massexpand. In the first run (see Table 8.2) BTHAI Terran lost all six games. After the modification of the buildorder and squads BTHAI Terran won 5 of 6 games. The modified buildorder and squads setup files are shown in Table 8.7 and Table 8.8. BTHAI was also the only (from our knowledge) bot from the AIIDE 2010 competition that could play all three races.

A common denominator for the top bots (BTHAI was not part of this group) in the experiment was a very effective micro-management of groups of units. For example Overmind, the winner of the AIIDE 2010 tournament, sends workers to steal gas from the opponent player. Chronos uses a very effective hit-and-run tactic with groups of cloaked Terran Wraiths. The use of potential fields makes our units effective when engaging the enemy by surrounding his/her units in a half circle. There are however other things that can be improved regarding micro-management, for example to use flying units to take out enemy workers instead of attacking the front of a base, try gas steal like Overmind, be better at guarding Siege Tanks, use special abilities more effectively, and more. The inheritance trees used for unit agents, structure agents and squads makes it easy to modify or create new logic for units or squads. A new type of KiteSquad can be added that locates enemy supply lines and tries to take them out, MarineAgent can be modified to be better at assisting important Siege Tanks under attack, and more. One example already implemented is the ReaverBaseHarass squad that uses a Protoss Shuttle to drop slow-moving but strong Protoss Reavers in the enemy base.

Another thing is adaptivity, i.e. to eventually change tactics depending on what the opponent does. BTHAI can have different buildorders, upgrades, techs and squads setup depending on the race of the opponent but besides that no adaption of tactics are performed at runtime. Currently the *Commander* agent is very simple; when all required squads are completed it launches an attack against the enemy. The *Commander* also sends squads to assist bases under attack. It is easy to add a new *Commander* agent extending the current one to create more complex logic and adaptivity.

We believe that the main goals of creating a general, expandable bot that can be used for future research and education in RTS games are fulfilled. The bot can play reasonably well in terms of winning games, we have shown that it is easy to modify or add new logic, and the use of text files for squads and buildorders makes it easy to change and test new tactics.

8.6 Future Work

The buildorders/upgrades/techs/squads setup text files used in the experiments are manually created based on our own knowledge about StarCraft. Weber and Mateas used a data mining tool to extract tactical information from replays of human experts playing StarCraft, and used that information to detect the opponent's strategy (Weber & Matteas, 2009). Synnaeve and Bessière used the dataset created by Weber and Mateas to predict opening tactics of opponents (Synnaeve & Bessiere, 2011). A similar approach could be used in BTHAI. Tactical information could be gatherered by automatically analyze replays or extracted from Weber and Mateas' database, and that information could then be used to automatically generate text files for all player/opponent combinations. Another interesting approach could be to use genetic programming to evolve tactics. This is easy to do in BTHAI since the tactics are separated from the bot code and generated tactics file can be tested without any recompilation needed.

Currently BTHAI uses one text file for each player/opponent combination. It could also be possible to have several text files for each combination, and let the bot choose one at the start of each game. The tactics to use could either be chosen randomly or selected based on knowledge of opponent and map features. Learning could also be used to find the tactics that works best against a specific opponent, but that will require a larger number of games played against each opponent than in the experiments described in this paper.

The *ExplorationManager* in BTHAI gathers intelligence data from scouting units, for example where the opponent has placed defensive structure or what types of units he/she prefers. Currently this data is not used, and a future improvement could be for example to use the data to locate weak spots in the opponents' defenses and launch surprise attacks at those spots. It could also be possible to add or modify own squads setup to counter the tactics used by the enemy. This will probably require a more complex script language for squads setup files that can handle selection cases.

8.7 Appendix

Terran_Supply_Depot	Terran_Supply_Depot
Terran_Barracks	Terran_Factory
Terran_Refinery	Terran_Missile_Turret
Terran_Academy	Terran_Armory
Terran_Factory	Terran_Command_Center
Terran_Command_Center	Terran_Supply_Depot
Terran_Supply_Depot	Terran_Missile_Turret
Terran_Bunker	Terran_Starport
Terran_Barracks	Terran_Science_Facility
Terran_Supply_Depot	Terran_Bunker
Terran_Command_Center	Terran_Missile_Turret
Terran_Engineering_Bay	Terran_Starport
Terran_Missile_Turret	Terran_Command_Center
Terran_Missile_Turret	

Table 8.4: *Buildorder file for BTHAI Terran.*

U-238_Shells:1	Terran_Infantry_Armor:1
Tank_Siege_Mode:1	Terran_Vehicle_Plating:1
Terran_Infantry_Weapons:1	Terran_Vehicle_Plating:1
Terran_Infantry_Weapons:1	Terran_Vehicle_Plating:1
Terran_Infantry_Weapons:1	Terran_Ship_Weapons:2
Stim_Packs:1	Terran_Ship_Weapons:2
Ion_Thrusters:1	Terran_Ship_Weapons:2
Terran_Vehicle_Weapons:1	Cloaking_Field:2
Terran_Vehicle_Weapons:1	EMP_Shockwave:2
Terran_Vehicle_Weapons:1	Irradiate:2
Terran_Infantry_Armor:1	Yamato_Gun:2
Terran_Infantry_Armor:1	

Table 8.5: *Upgrades file for BTHAI Terran.*

```
<start>
Type=Offensive
Move=Ground
Name=MainAttackSquad
Priority=10
ActivePriority=10
OffenseType=Required
<setup>
Unit=Terran_Marine:10
Unit=Terran_Medic:3
Unit=Terran_Goliath:3
Unit=Terran_Siege_Tank_Tank_Mode:3
<end>

<start>
Type=Offensive
Move=Ground
Name=MainAttackSquad
Priority=10
ActivePriority=10
OffenseType=Required
<setup>
Unit=Terran_Marine:10
Unit=Terran_Medic:3
Unit=Terran_Goliath:3
Unit=Terran_Siege_Tank_Tank_Mode:3
<end>

<start>
Type=Offensive
Move=Ground
Name=TankSquad
Priority=10
ActivePriority=10
OffenseType=Required
<setup>
Unit=Terran_Siege_Tank_Tank_Mode:5
<end>

<start>
Type=Exploration
Move=Air
Name=AirExplorerSquad
Priority=10
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Terran_Wraith:1
<end>
```

```
<start>
Type=Support
Move=Air
Name=ScienceVesselSquad
Priority=10
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Terran_Science_Vessel:1
<end>

<start>
Type=Support
Move=Air
Name=ScienceVesselSquad
Priority=10
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Terran_Science_Vessel:1
<end>

<start>
Type=Kite
Move=Air
Name=AirSquad
Priority=10
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Terran_Wraith:5
<end>

<start>
Type=Offensive
Move=Air
Name=HeavyAirSquad
Priority=10
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Terran_Battlecruiser:2
<end>
```

Table 8.6: *Squads setup file for BTHAI Terran.*

Terran_Supply_Depot	Terran_Factory
Terran_Barracks	Terran_Supply_Depot
Terran_Refinery	Terran_Armory
Terran_Bunker	Terran_Command_Center
Terran_Academy	Terran_Missile_Turret
Terran_Factory	Terran_Missile_Turret
Terran_Supply_Depot	Terran_Starport
Terran_Bunker	Terran_Science_Facility
Terran_Barracks	Terran_Missile_Turret
Terran_Engineering_Bay	Terran_Starport

Table 8.7: *Modified buildorder file for BTHAI Terran.*

<start>	<start>
Type=Offensive	Type=Support
Move=Ground	Move=Air
Name=MainAttackSquad	Name=ScienceVesselSquad
Priority=9	Priority=10
ActivePriority=10	ActivePriority=10
OffenseType=Required	OffenseType=Optional
<setup>	<setup>
Unit=Terran_Marine:20	Unit=Terran_Science_Vessel:1
Unit=Terran_Medic:6	<end>
Unit=Terran_Goliath:5	
Unit=Terran_Siege_Tank_Tank_Mode:6	<start>
<end>	Type=Support
	Move=Air
<start>	Name=ScienceVesselSquad
Type=Offensive	Priority=10
Move=Ground	ActivePriority=10
Name=TankSquad	OffenseType=Optional
Priority=10	<setup>
ActivePriority=10	Unit=Terran_Science_Vessel:1
OffenseType=Optional	<end>
<setup>	
Unit=Terran_Siege_Tank_Tank_Mode:5	<start>
<end>	Type=Offensive
	Move=Air
<start>	Name=AirSquad
Type=Exploration	Priority=10
Move=Air	ActivePriority=10
Name=AirExplorerSquad	OffenseType=Optional
Priority=10	<setup>
ActivePriority=10	Unit=Terran_Wraith:5
OffenseType=Optional	Unit=Terran_Battlecruiser:2
<setup>	<end>
Unit=Terran_Wraith:1	
<end>	

Table 8.8: *Modified squads setup file for BTHAI Terran.*

Protoss_Pylon	Protoss_Robotics_Support_Bay
Protoss_Gateway	Protoss_Pylon
Protoss_Assimilator	Protoss_Gateway
Protoss_Cybernetics_Core	Protoss_Photon_Cannon
Protoss_Citadel_of_Adun	Protoss_Gateway
Protoss_Templar_Archives	Protoss_Stargate
Protoss_Gateway	Protoss_Arbiter_Tribunal
Protoss_Nexus	Protoss_Gateway
Protoss_Forge	Protoss_Photon_Cannon
Protoss_Photon_Cannon	Protoss_Nexus
Protoss_Robotics_Facility	Protoss_Stargate
Protoss_Observatory	Protoss_Photon_Cannon
Protoss_Nexus	Protoss_Fleet_Beacon

Table 8.9: *Buildorder file for BTHAI Protoss.*

Protoss_Ground_Weapons:1	Protoss_Ground_Armor:2
Leg_Enhancements:1	Protoss_Ground_Armor:2
Protoss_Ground_Weapons:1	Scarab_Damage:2
Psionic_Storm:1	Protoss_Air_Weapons:3
Singularity_Charge:1	Protoss_Air_Weapons:3
Khaydarin_Amulet:1	Protoss_Air_Weapons:3
Protoss_Ground_Weapons:1	Apial_Sensors:3
Reaver_Capacity:2	Gravitic_Thrusters:3
Protoss_Ground_Armor:2	Carrier_Capacity:3

Table 8.10: *Upgrades file for BTHAI Protoss.*

<start>	<start>
Type=Offensive	Type=ReaverBaseHarass
Move=Ground	Move=Air
Name=MainAttackSquad	Name=ReaverSquad
Priority=9	Priority=10
ActivePriority=10	ActivePriority=10
OffenseType=Required	OffenseType=Optional
<setup>	<setup>
Unit=Protoss_Zealot:6	Unit=Protoss_Shuttle:1
Unit=Protoss_Dragoon:5	Unit=Protoss_Reaver:2
<end>	<end>
<start>	<start>
Type=Rush	Type=Exploration
Move=Ground	Move=Air
Name=RogueSquad	Name=AirExplorerSquad
Priority=10	Priority=10
ActivePriority=10	ActivePriority=10
OffenseType=Optional	OffenseType=Optional
<setup>	<setup>
Unit=Protoss_Dark_Templar:6	Unit=Protoss_Observer:1
<end>	<end>
<start>	<start>
Type=Offensive	Type=Support
Move=Ground	Move=Air
Name=MainAttackSquad	Name=SupportSquad
Priority=10	Priority=10
ActivePriority=10	ActivePriority=10
OffenseType=Required	OffenseType=Optional
<setup>	<setup>
Unit=Protoss_Zealot:6	Unit=Protoss_Arbiter:3
Unit=Protoss_Dragoon:10	<end>
Unit=Protoss_High_Templar:2	
Unit=Protoss_Observer:1	<start>
<end>	Type=Offensive
	Move=Air
<start>	Name=AirSquad
Type=Offensive	Priority=10
Move=Ground	ActivePriority=10
Name=MainAttackSquad	OffenseType=Optional
Priority=10	<setup>
ActivePriority=10	Unit=Protoss_Scout:5
OffenseType=Required	<end>
<setup>	
Unit=Protoss_Zealot:6	<start>
Unit=Protoss_Dragoon:10	Type=Offensive
Unit=Protoss_High_Templar:2	Move=Air
Unit=Protoss_Observer:1	Name=AirSquad
<end>	Priority=10
	ActivePriority=10
	OffenseType=Required
	<setup>
	Unit=Protoss_Corsair:3
	<end>

Table 8.11: *Squads setup file for BTHAI Protoss.*

Zerg_Spawning_Pool	Zerg_Evolution_Chamber
Zerg_Extractor	Zerg_Hatchery
Zerg_Hatchery	Zerg_Queens_Nest
Zerg_Hydralisk_Den	Zerg_Defiler_Mound
Zerg_Spire	Zerg_Hatchery
Zerg_Hatchery	Zerg_Creep_Colony
Zerg_Creep_Colony	Zerg_Sunken_Colony
Zerg_Sunken_Colony	Zerg_Hatchery
Zerg_Hatchery	Zerg_Creep_Colony
Zerg_Creep_Colony	Zerg_Sunken_Colony
Zerg_Sunken_Colony	

Table 8.12: *Buildorder file for BTHAI Zerg.*

Lurker_Aspect:1	Zerg_Missile_Attacks:2
Antennae:2	Zerg_Missile_Attacks:2
Pneumatized_Carapace:2	Zerg_Flyer_Attacks:2
Zerg_Carapace:3	Zerg_Flyer_Attacks:2
Zerg_Carapace:3	Zerg_Flyer_Attacks:2
Zerg_Carapace:3	Consume:3
Zerg_Missile_Attacks:2	

Table 8.13: *Upgrades file for BTHAI Zerg.*

```
<start>
Type=Exploration
Move=Air
Name=ExplorationSquad
Priority=5
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Zerg_Overlord:1
<end>

<start>
Type=Rush
Move=Ground
Name=LurkerRushSquad1
Priority=8
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Zerg_Hydralisk:4
<end>

<start>
Type=Rush
Move=Ground
Name=LurkerRushSquad2
Priority=9
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Zerg_Hydralisk:4
<end>

<start>
Type=Offensive
Move=Air
Name=AirSquad
Priority=9
ActivePriority=10
OffenseType=Required
<setup>
Unit=Zerg_Mutalisk:12
<end>

<start>
Type=Offensive
Move=Ground
Name=LurkerSquad
Priority=10
ActivePriority=10
OffenseType=Required
<setup>
Unit=Zerg_Hydralisk:8
<end>
```

```
<start>
Type=Offensive
Move=Ground
Name=MainAttackSquad
Priority=10
ActivePriority=10
OffenseType=Required
<setup>
Unit=Zerg_Hydralisk:20
Unit=Zerg_Overlord:1
<end>

<start>
Type=Offensive
Move=Air
Name=DevourerSquad
Priority=10
ActivePriority=10
OffenseType=Required
<setup>
Unit=Zerg_Mutalisk:6
Unit=Zerg_Queen:2
<end>

<start>
Type=Offensive
Move=Air
Name=QueenSquad
Priority=10
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Zerg_Queen:2
<end>

<start>
Type=Offensive
Move=Ground
Name=DefilerSquad
Priority=10
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Zerg_Defiler:5
<end>

<start>
Type=Offensive
Move=Air
Name=ScourgeSuicideSquad
Priority=10
ActivePriority=10
OffenseType=Optional
<setup>
Unit=Zerg_Scourge:10
<end>
```

Table 8.14: *Squads setup file for BTHAI Zerg.*

CHAPTER

NINE

PAPER VIII

Measuring player experience on runtime dynamic difficulty scaling in an RTS game.

Johan Hagelbäck & Stefan J. Johansson

Proceedings of 2009 IEEE Symposium on Computational Intelligence and Games (CIG), 2009.

9.1 Introduction

The important thing is not winning but taking part!?
The saying origins from the Olympic Creed that once was formulated by the founder of the Olympic Committee, Pierre de Coubertin, in the beginning of the last century (Macaloon, 2007):

> The important thing in the Olympic Games is not winning but taking part.
> Just as in life, the aim is not to conquer but to struggle well.

These words are, as we will argue, indeed applicable also to computer games.

We have made a study on player experience of five different computer opponents in an RTS game, two with static difficulty setting and three which changes difficulty setting dynamically during a game. Traditionally difficulty settings of computer opponents in games are set manually and a player might reach a point where the opponents are no longer a challenge for him. Another issue can be that the challenge step between the pre-defined difficulty levels is too wide. A player might find a difficulty level too easy, but raising the level one step is a too big challenge for him. A third issue is that a player might discover a weakness in the computer opponent's tactics, and once discovered the player wins easily by exploiting the weakness.

The study was carried out during DreamHack Winter 2008, the largest LAN party in the world. The event is held yearly in Jönköping in Sweden and attracted more than 13.800 participants (DreamHack, 2009). The participants in our experiments played a game against one of the five bots in an RTS game and were asked to fill in a questionnaire after the game has ended. A total of 60 persons participated in the study.

9.1.1 Real Time Strategy Games

In real-time strategy, RTS, games the player has to construct and control an army of soldiers and vehicles and use it to destroy the opponent forces. Typically the player has to gather resources to construct buildings which in turn allows the player to build offensive and defensive units. The game runs in real-time in contrast to turn-based strategy games such as Civilization. Famous titles in the genre is Command & Conquer, Warcraft, StarCraft and Age of Empires.

9.1.2 Measuring Enjoyment in Games

There are several different models of player enjoyment in computer games, ranging from the work of Malone in the early 80's on intrinsic qualitative factors for engaging game play (Malone, 1981a, 1981b), to the work of e.g. Sweetster and Wyeth on the Gameflow model (Sweetster & Wyeth, 2005). We will in our paper not try to model the enjoyment as such, e.g by partition it into factors. Instead we let the players express their experience in terms of a number of adjectives of six different clusters. We then analyse the players' opinions about the enjoyment to their opinions about the strength and the variation of the computer opponent. This does not measure the enjoyment in any absolute way (and that is not our intention either), but relate it to properties of strength and variation.

9.1.3 Dynamic difficulty scaling

Difficulty scaling means that the difficulty of a game is adjusted to suit the skills of the human player. The purpose is to give the player a challenge even when his skill in the game increases. It is typically set manually by the player by choosing from different pre-set difficulty levels (e.g. Beginner, Normal, Hard, Expert). Another approach is to use built-in adaption of the difficulty in the game AI. Bakkes et al. describes an approach called *rapidly adaptive game AI* where the difficulty in the RTS game Spring is adapted at runtime by using observations of current game state to estimate the probable outcome of the game (Bakkes, Spronck, & Herik, 2008).

Olesen et al. describes an approach where several factors that contribute to the difficulty were identified, and artifical neural network controlled agents that excel on those factors were trained offline and used in dynamic difficulty scaling (Olesen, Yannakakis, & Hallam, 2008).

Both approaches uses an evaluation function to estimate the current relative strength between the player and the AI.

9.1.4 Outline

We will first go through the environment which we will use to conduct the experiments, followed by a short description of the bot that we use and the modifications made to it. In Section 9.3 we will describe the experimental setup and the results are presented in Section 9.4. We finish by discussing the results and the methodology, drawing conclusions and line out directions for future work.

9.2 The Open Real Time Strategy Platform

Open Real Time Strategy (ORTS) (Buro, 2007a) is an open source real-time strategy game engine developed by the University of Alberta as a tool and test-bed for real-time AI research. Through an extensive scripting system the game engine supports many different types of games. In the experiments performed in this paper we used the Tankbattle scenario. In Tankbattle each player start with five bases and 50 tanks each. The goal is to destroy all the bases of the opponent. The number of units and bases is fixed and no additional units can be constructed during the game.

9.2.1 Multi-Agent Potential Fields

The Multi-Agent Potential Field based bot used in the experiments is based on previous work we conducted in Papers I and III. The idea is to generate potential fields by placing attracting or repelling affectors at interesting positions in the game world, for example enemy units and buildings. The different fields are weighted and summed together to form a total potential field which is used by the agents, tanks, for navigation.

We identified four tasks in the Tankbattle scenario: Avoid colliding with moving objects, Hunt down the enemy's forces, Avoid colliding with cliffs, and Defend the bases. Three major types of potential fields are used to handle the tasks: *Field of Navigation*, *Strategic Field*, and *Tactical field*.

The field of navigation is generated by letting every static terrain tile generate a small repelling force. We would like our agents to avoid getting too close to objects where they may get stuck, but instead smoothly pass around them.

The strategic field is an attracting field. It makes agents go towards the opponents and place themselves at appropriate distances from where they can fight the enemies. The field is made up from subfields generated by all enemy tanks and bases. The generated subfields are symmetric with the highest, i.e. most attractive, potentials in a circle located at Maximum Shooting Distance (MSD) from the enemy unit or structure. The reason for placing the most attractive potential at MSD is that we want our tanks to surround and fight the enemy from the max distance of their cannons instead of engaging the enemy in close combat. This is illustrated in Figure 9.1. It shows an own tank attacking an enemy unit from maximum shooting distance.

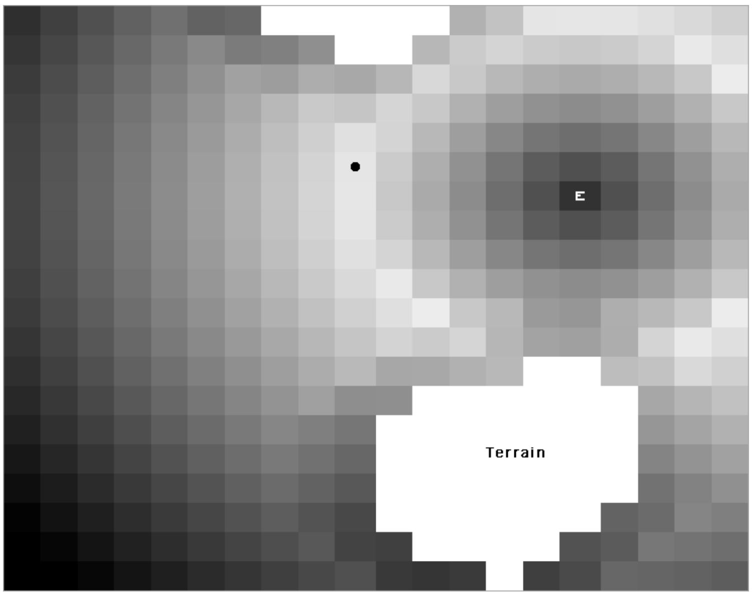


Figure 9.1: *An own tank (black circle) engaging an enemy unit E. The most attractive potentials are in a circle surrounding the enemy at maximum shooting distance of the tank.*

The tactical field is generated by own units, own bases and sheep. These objects generate small repelling fields to avoid our agents from colliding with each other or bases as well as avoiding the sheep.

Each subfield is weighted and summed to a major field, and the major fields are in turn weighted and summed to form a total potential field which is used by our agents for navigation. We will illustrate how the total potential field can look like with an example. Figure 9.2 shows a potential field view of a worker unit moving from a base to a mine to gather resources. The mine is the goal of the unit and therefore generates an attracting field (lighter grey areas have higher potentials than darker grey areas). The unit must also avoid colliding with obstacles, and the base and terrain therefore generate small repelling fields.

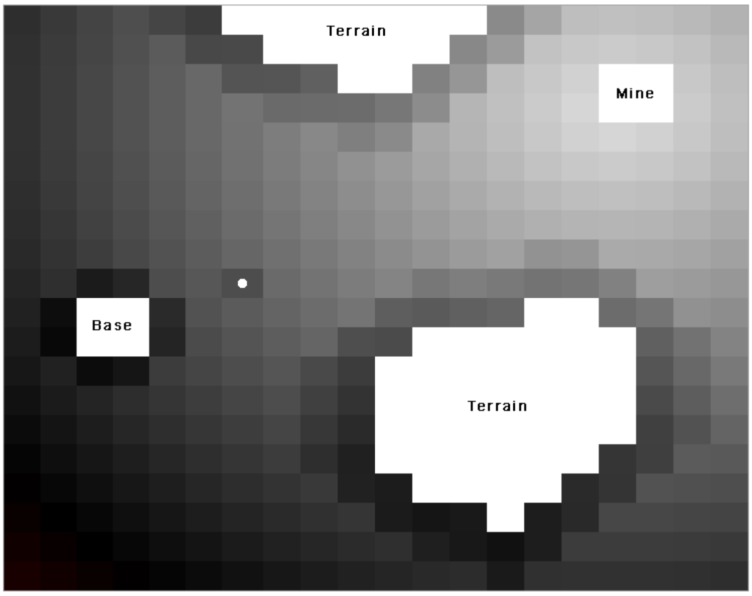


Figure 9.2: *A worker unit (white circle) moving towards a mine to gather resources. The mine generates an attractive field and the base and terrain generate small repelling fields for obstacle avoidance. Light grey areas are more attracting than darker grey areas. White areas are impassable tiles.*

Figures 9.3 and 9.4 shows a 2D debug view from the game server and the corresponding potential field view during an ORTS tankbattle game. It illustrates how our own units cooperate to engage and surround the enemy at maximum shooting distance. For the interested reader we refer to the original description for more details of the MAPF technology in Papers I and III.

9.3 Experimental Setup

The experiments were carried out during DreamHack Winter 2008, the largest LAN party in the world. We were positioned in the boot of our University, where players who stopped at our boot were asked if they would like to participate in a scientific experiment. Those who agreed were given instructions on how to play the game and then played a short session of *ORTS Tank Battle* against one of the bots. After the match, the participants filled in a questionnaire.

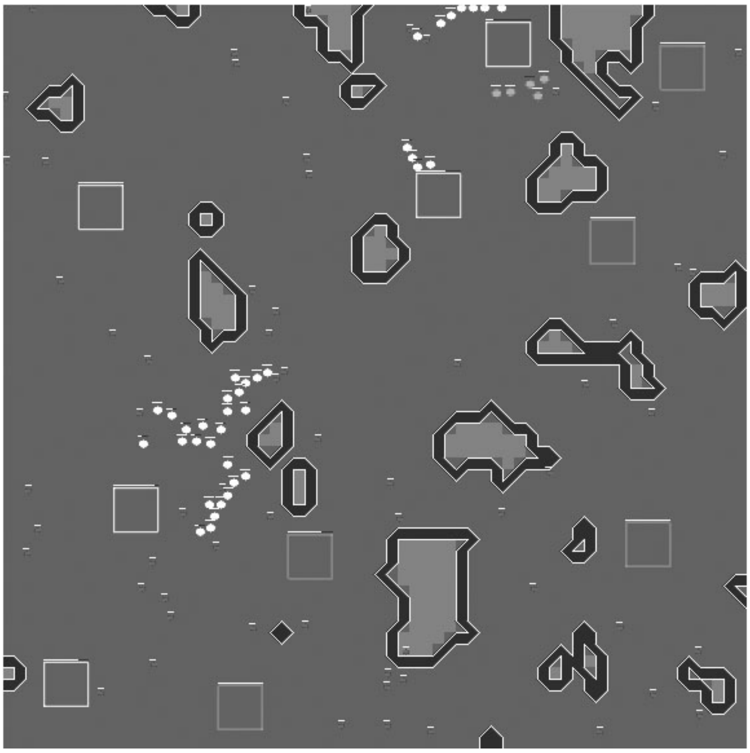


Figure 9.3: *The 2D view of an ORTS Tankbattle game. Our own tanks (white circles) are engaging the enemy units (grey circles) and bases (grey rectangles).*

9.3.1 The Different Bots

We used five different bots, each one based on the MAPF-bot described in Paper III. The only changes made were that each unit's ability to act in each time frame was restricted through a probabilistic filter in the following way:

- A bot strength, s, is initialised in the beginning of the game.
- In each time frame and for each unit, generate a random number $r \in [0, 1]$.
- If $r > s$, let the unit be idle for this time frame.
- If $r \leq s$, let the unit act as usual.

The only thing that we varied in our experiments was the value of s. Two of the bots used constant values of s, while three versions used dynamically adjusted values. The

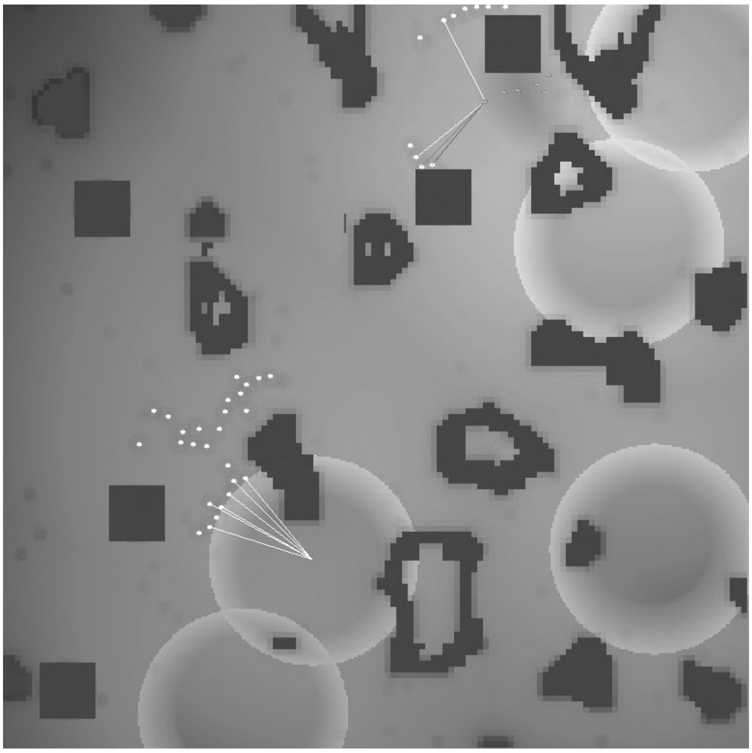


Figure 9.4: *The potential field view of the same ORTS Tankbattle game. Lighter grey areas have more attracting potentials than darker areas. The white lines illustrate the coordinated attacks on a base (lower left) and a unit (upper right).*

five bot versions are:

Bot A

Static bot with medium difficulty. s is set to 0.3 and does not change during the game.

Bot B

Static bot with low difficulty. s is set to 0.1 and does not change during the game.

Bot C

Adaptive bot. s is initially set to 0.4 and is adjusted during the game using the adaptive difficulty algorithm described below. Learningrate is set to 0.01.

Bot D

Same as Bot C, but when the human player has five or less tanks left s drops to 0.02 to always let the player win.

Bot E

Adaptive bot with high learningrate to quickly adapt to changes in the game. s is initially set to 0.4 and is updated with learningrate 0.2.

9.3.2 Adaptive difficulty algorithm

In the adaptive bot versions s is adjusted throughout the course of a game using an adaptive difficulty scaling algorithm. An evaluation function is first used to calculate the relative strength, $isScore$, between the computer and human player at a given time frame. A score above 0 means that the human player is in lead, and below 0 the computer player is in lead. The $aimScore$ is the relative strength the bot aims for, in our experiments 0. By changing the $aimScore$ the difficulty of the adaptive AI is adjusted. By, for example, setting the value to slightly above 0 the goal of the difficulty scaling algorithm is to let the human player be in a small lead. A high positive value will in turn create a computer opponent that is easily beaten. The approach of observing the relative strength between the players were also used by Bakkes et al. in (Bakkes et al., 2008). The difference compared to our approach is in the features involved in the calculation of the relative strength.

The pseudocode below describes how the evaluation score is used to update s. Note that the implementation makes sure s always is between 0.05 and 1.0, except for late in a game for Bot D where s is set to 0.02 bypassing the algorithm. The adaptive difficulty algorithm was first evaluated against two static bots from the 2007 years' annual ORTS tournament. The purpose was to make sure the difficulty scaling worked correctly before performing experiments with human players. The results from the evaluation is presented in Table 9.1. They show that the scaling, although not entirely perfect, works well enough. The adaptive difficulty bot does not have to win exactly 50% of the games. The goal is rather that it reacts to the opponent and tries to play an even game, which is confirmed by the experiments.

Algorithm 1 The algorithm that updates the s parameter.

```
$isScore = 0$
$aimScore = 0$
**for all** EnemyUnit $eu$ **do**
  $isScore = isScore + 1 + eu.getHealthPercent() * 0.5$
**end for**
**for all** OwnUnit $ou$ **do**
  $isScore = isScore - 1 - ou.getHealthPercent() * 0.5$
**end for**
$diffScore = isScore - aimScore$
$s = s + learningRate * diffScore$
**if** $s < 0.05$ **then**
  $s = 0.05$
**end if**
**if** $s > 1.0$ **then**
  $s = 1.0$
**end if**
```

Table 9.1: *Win percent of our bot, both with and without adaptive difficulty scaling, against two bots from 2007 years' ORTS tournament.*

	Wins (over 100 games)	
Opponent	No Scaling	Adaptive Difficulty
NUS	100%	60%
Uofa06	100%	52%

9.3.3 The Questionnaire

The Questionnaire consisted of a brief instruction on how to fill in the answers, a number of general questions (concerning age, sex, and the results of the match, which were filled in by the test leader), and a word pair section.

Before the creation of the questionnaire, we identified three aspects of the opponent bot that we would like to test:

- The experienced *enjoyment* of facing this opponent bot,
- the experienced *strength* of the opponent bot, and
- the experienced *variation* of the opponent bot.

We used six clusters of words, with four words in each cluster (in total 24 words) to describe the bot along three different dimensions. In the questionnaire, the participants had to compare twelve pairs of words, where words from each cluster was paired with words from each other cluster except the opposite one. *Hard, Demanding, Challenging,* and *Impossible* was for instance compared to *Fun, Painful, Monotonous* and *Alternating*, rather than the words in the cluster of *Uncomplicated, Easy, Simple* and *problem free*. One can think of it as the six clusters being the six sides of a cube. Each side shares four edges, on with every other side, except the opposite side. Each edge at that cube then correspond to a pair of words in our experiment.

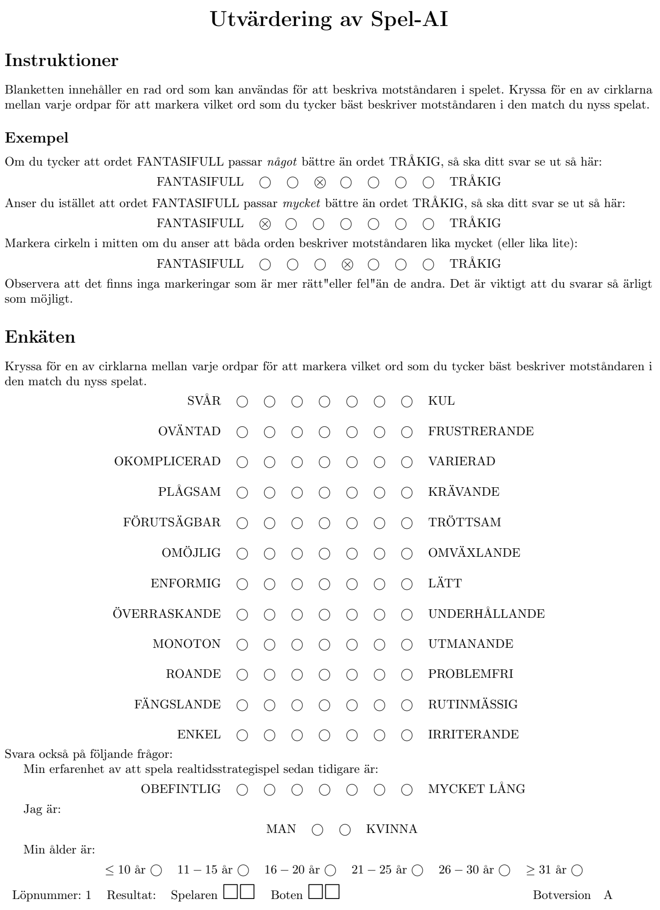
Each comparison as such was carried out by the user through marking one of seven options along an ordinal scale, where the center option indicated that both words described the bot equally good (or bad), and the others ranged from *somewhat better* to *much better* in favour of the closest word.

The words used for the different axis are presented in Table 9.2. Since the experiments were carried out in Sweden, we used a Swedish questionnaire (therefore the original words of the questionnaire are provided as well).

Table 9.2: *A list of the words used in the questionnaire. Word 1 and Word 2 of the same row in the table were also compared in the questionnaire.*

Word 1	in Swedish	Word 2	in Swedish
Hard	Svår	Fun	Kul
Unexpected	Oväntad	Frustrating	Frustrerande
Uncomplicated	Okomplicerad	Variated	Varierad
Painfull	Plågsam	Demanding	Krävande
Predictable	Förutsägbar	Tiresome	Tröttsam
Impossible	Omöjlig	Alternating	Omväxlande
Tedious	Enformig	Easy	Lätt
Surprising	Överraskande	Entertaining	Underhållande
Monotonous	Monoton	Challenging	Utmanande
Enjoying	Roande	Problem free	Problemfri
Captivating	Fängslande	Routine	Rutinmässig
Simple	Enkel	Irritating	Irriterande

The participants did of course not know what version of the bot they faced in their experiment, but we did inform them that it was an experiment aimed at trying to measure different experienced properties of the computer opponent. In total 60 persons participated, evenly distributed among the five different versions of the bot.



Utvärdering av Spel-AI

Instruktioner

Blanketten innehåller en rad ord som kan användas för att beskriva motståndaren i spelet. Kryssa för en av cirklarna mellan varje ordpar för att markera vilket ord som du tycker bäst beskriver motståndaren i den match du nyss spelat.

Exempel

Om du tycker att ordet FANTASIFULL passar *något* bättre än ordet TRÅKIG, så ska ditt svar se ut så här:

FANTASIFULL ○ ○ ⊗ ○ ○ ○ ○ TRÅKIG

Anser du istället att ordet FANTASIFULL passar *mycket* bättre än ordet TRÅKIG, så ska ditt svar se ut så här:

FANTASIFULL ⊗ ○ ○ ○ ○ ○ ○ TRÅKIG

Markera cirkeln i mitten om du anser att båda orden beskriver motståndaren lika mycket (eller lika lite):

FANTASIFULL ○ ○ ○ ⊗ ○ ○ ○ TRÅKIG

Observera att det finns inga markeringar som är mer rätt"eller fel"än de andra. Det är viktigt att du svarar så ärligt som möjligt.

Enkäten

Kryssa för en av cirklarna mellan varje ordpar för att markera vilket ord som du tycker bäst beskriver motståndaren i den match du nyss spelat.

SVÅR	○	○	○	○	○	○	○	KUL
OVÄNTAD	○	○	○	○	○	○	○	FRUSTRERANDE
OKOMPLICERAD	○	○	○	○	○	○	○	VARIERAD
PLÅGSAM	○	○	○	○	○	○	○	KRÄVANDE
FÖRUTSÄGBAR	○	○	○	○	○	○	○	TRÖTTSAM
OMÖJLIG	○	○	○	○	○	○	○	OMVÄXLANDE
ENFORMIG	○	○	○	○	○	○	○	LÄTT
ÖVERRASKANDE	○	○	○	○	○	○	○	UNDERHÅLLANDE
MONOTON	○	○	○	○	○	○	○	UTMANANDE
ROANDE	○	○	○	○	○	○	○	PROBLEMFRI
FÄNGSLANDE	○	○	○	○	○	○	○	RUTINMÄSSIG
ENKEL	○	○	○	○	○	○	○	IRRITERANDE

Svara också på följande frågor:

Min erfarenhet av att spela realtidsstrategispel sedan tidigare är:

OBEFINTLIG ○ ○ ○ ○ ○ ○ ○ MYCKET LÅNG

Jag är:

MAN ○ ○ KVINNA

Min ålder är:

≤ 10 år ○ $11 - 15$ år ○ $16 - 20$ år ○ $21 - 25$ år ○ $26 - 30$ år ○ ≥ 31 år ○

Löpnummer: 1 Resultat: Spelaren ☐☐ Boten ☐☐ Botversion A

Figure 9.5: *The questionnaire used in the experiments. In addition to the instructions, an example of how to fill it in, and the word pair questions, we also asked about the player's age and sex, and asked them to grade their previous experience from RTS games.*

Table 9.3: *The clustering of words in the six clusters.*

Enjoyment+			
Entertaining	Fun	Enjoying	Attractive
Enjoyment-			
Painfull	Tiresome	Tedious	Irritating
Strength+			
Hard	Demanding	Impossible	Challenging
Strength-			
Uncomplicated	Easy	Problem free	Simple
Variation+			
Unexpected	Variated	Alternating	Surprising
Variation-			
Frustrating	Predictable	Monotonous	Routine

9.4 The Results

Since the options in each decision in the questionnaire were formulated in terms of which word that best describes the bot, we do not take into account the possibility that a participant use the scale to negate a property by giving high marks to the other word. Only the cases where the participant decide in favour of a word will count for that cluster, as illustrated in Table 9.4.

Table 9.4: *The distribution of scores given a participant mark comparing* Hard *and* Fun. *The scores as shown below in the lower two rows were not revealed to the players in the questionnaire.*

HARD	○	○	○	○	○	○	○	FUN
Strength+	3	2	1	0	0	0	0	
Enjoyment+	0	0	0	0	1	2	3	

We now compare the different aspects A of each bot B as expressed by the participants P in the following way:

$$B_A = \frac{\sum_{\substack{p \in P \\ p \text{ tested } B}} \sum_{i=1}^{4} ((A+)_i - (A-)_i)}{|\{p \text{ tested } B\}|} \tag{9.1}$$

I.e we took the average scores for each aspect for each and one of the bots. The results are presented in Table 9.5.

Table 9.5: *The average results of the three aspects.*

Bot	Enjoyment	Strength	Variation
A	-0.03	0.46	-0.69
B	-0.04	-1.6	-0.64
C	0.66	-0.38	-0.03
D	-0.94	-1.21	-1.17
E	0.91	-0.62	-0.17

9.5 Discussion

9.5.1 The Methodology

We chose to let the participants only try *one* of the five versions. This choice can of course be criticised, since it has been argued that experiments where you let the participants compare a number of different options give more reliable results (Yannakakis & Hallam, 2007). Even though they indeed have a point, especially for calmer environments where the participants have reserved time for the experiments, we were not at all convinced that we would be able to catch their interest so that they would play two or more games in this noisy, distractive environment.

9.5.2 The Results

Table 9.5 and 9.6 show that the adaptive Bots C and E were perceived as most varied. The third adaptive bot, D, had the lowest variation score of all. This bot aims for a balanced game until the end where the performance drops so the own tanks in practice are idle. The low variation score is probably due to that players have a tendency to remember the last parts of the game when filling in the forms. Since the bot is very dumbed down during this phase of the game it is perceived as static and boring. Mat Buckland's saying that "it only takes a small dodgy-looking event to damage a player's confidence in the AI" is very applicable to this bot version (Buckland, 2005).

The static Bot A was the most difficult to beat. The low performance static Bot B and the adaptive Bot D that drops performance were very easy to beat. The adaptive Bots C and E were somewhere in the middle with some participants perceiving them as relatively easy to beat while others felt they were quite hard to beat, which of course was the intention of using runtime difficulty scaling algorithm.

The adaptive Bots C and E were by far considered most enjoying to play against. The static Bots A and B were scored as enjoying by some participants and not enjoying by others, ending up around a zero score. A majority of the participants that played

Table 9.6: *The average results of the six clusters.*

Bot		E+	E-	S+	S-	V+	V-
A	Avg.	1.33	-1.24	1.65	-1.36	1.42	-1.75
	Stdd.	0.62	0.56	0.61	0.67	0.67	0.85
	n	15	17	17	11	12	24
B	Avg.	1.33	-1.55	1	-2	1.44	-1.79
	Stdd.	0.65	0.82	0	0.94	0.73	0.86
	n	12	11	4	26	16	29
C	Avg.	1.71	-1.64	1.38	-1.81	1.88	-1.57
	Stdd.	0.81	0.67	0.51	0.98	0.86	0.68
	n	24	11	13	16	17	21
D	Avg.	1.5	-2.27	1.33	-2.35	1.22	-2
	Stdd.	0.84	0.65	0.5	0.75	0.44	0.75
	n	6	11	9	20	9	26
E	Avg.	1.71	-1.5	1.42	-1.60	1.75	-1.53
	Stdd.	0.86	0.76	0.67	0.65	0.75	0.72
	n	24	8	12	25	12	17

against Bot D, felt that it was not fun at all to meet.

To further backup the findings we have performed a t-test with alpha level 0.02. The results from the test is presented in Table 9.7.

It shows that Bot A is perceived as difficult and not very varied, but we cannot say anything for sure about the perceived entertainment. The participants thought Bot B was very easy and not varied, but there were no significance in the entertainment results. Bot C was considered very entertaining, but with unsure difficulty and varation. Bot D was by a majority of the participants considered as boring, not very varied and not fun to play against. Bot E was easy, not very varied but fun to play against.

9.6 Conclusions and Future Work

The results show that the adaptive Bots C and E were perceived as comparingly varied, neither being too hard nor too easy to beat, and being the most fun to play against of the versions used in the experiment. This confirm our beliefs that players prefer opponents that models and adapts to the player and that can be beaten if the player plays the game well. It would however be interesting to see how adaptive bots with higher and lower difficulty levels would be perceived.

The static bots were, compared to the adaptive ones, perceived as boring to play

Table 9.7: *T-test for Strength, Variation and Entertainment for each bot.*

	S	V	E	t_S	t_V	t_E
Bot A	0.5	-1	-0,17	16.72	-16.72	-4.37
Bot B	-1.83	-1.08	0.08	-49.89	-22.8	2.64
Bot C	0	-0.25	1.33	0	-4.77	28.14
Bot D	-1.08	-1.42	-0.33	-24.58	-27.93	-11.26
Bot E	-1.08	-0.42	1.33	-25.28	-11.51	35.60

against and their difficulty level was not balanced to the player (Bot A was too hard and Bot B too easy to beat). It is often difficult to find a static difficulty level that suit the preferences of many players. The risk with static levels is that one difficulty setting is way too easy for a player while the next level is too hard. For example Warcraft III has been criticised in forums and reviews for being too difficult in skirmish games against computer players (Warcraft III Guide, 2008; Warcraft III Review, 2008). The adaptive Bot D that drops the performance in the end of a game was perceived as boring, easy to win against and with low variation. It is interesting to see that this bot was considered least varied even if it uses the same adaptive technique as Bot C until the performance drops and tanks in practice are almost idle. This might be because participants have a tendency to mostly remember the last part of the game where the AI is really dumb, and therefore consider the bot as too easy and boring.

Future work would include experiments in a more calm and controlled environment. The participants would get some time to play against a default opponent to learn a bit about the game mechanics, and then get to play against one of the bot versions. The different versions could also be slightly revised. Bot D, the one that drops performance considerably in the end game, is by the participants regarded as not very enjoyable, easy to beat and with least varation.

Another direction could be to use other techniques for compensating the difference in strength between the players. The approach we used was to let units be idle with some probability. Instead the difference can be compensated with changes in higher level tactics, for example switch to a more non-aggresive playing style if the computer player is in lead (Bakkes et al., 2008). Still it is interesting to see that such a simple way of runtime difficulty scaling was clearly perceived as more varied and entertaining than the static bots in the experiments.

CHAPTER
TEN

PAPER IX

A Study on Human like Characteristics in Real Time Strategy Games.

Johan Hagelbäck & Stefan J. Johansson
Proceedings of 2010 IEEE Conference on Computational Intelligence and Games (CIG), 2010.

10.1 Introduction

Games contain a large amount of computer controlled characters (we will in the rest of the paper use the abbreviation NPCs, non-player characters). They can be opponents the players has to beat, team members the player has to cooperate with, or inhabitants in cities which the player might or might not be able to interact with. The purpose of computer controlled characters is to populate the game world and give more depth and playability to the game. This is especially important in single-player games where the computer controlled characters should give the player a challenge but not be overpowered, and make the game fun to play. Because that is why we play games, for fun!

10.1.1 Humanlike NPCs

The goal is often to create interesting NPCs by making them more humanlike. It is debatable if a more humanlike NPC is more fun to play against. Yannakakis and Hallam mentions in (Yannakakis & Hallam, 2007) that this does not always have to be the case. Soni and Hingston created humanlike bots in the first-person shooter (FPS) game Unreal Tournament by letting human players train bots how to behave using neural networks. Their experiments showed that the bots trained to behave more humanlike were clearly perceived as more fun (and humanlike) than the coded bots (Soni & Hingston, 2008). Freed et.al. has made a survey to identify the significant differences between human and computer play in the RTS game StarCraft. They conclude that more humanlike bots can be valuable for training new players in a game as well as providing the players with a better testing ground for new tactical and strategic ideas (Freed et al., 2007). They also mention that one of the main problems when creating such bots is to understand the significant aspects of how humans play a game.

We are not really at the point where we can create actual intelligence, so we have to focus on what Scott defines as "The Illusion of Intelligence" (Scott, 2002). The NPCs does not have to be intelligent, as long as they do reasonably intelligent actions under most circumstances. What is reasonably intelligent and humanlike is of course very genre specific. A soldier sneaking through a jungle full of enemies has to take very different decisions than a basketball player in the middle of an exciting match.

10.1.2 In this paper

We have performed a study where participants were asked to observe a replay of a Real-Time Strategy (RTS) game. The replays were created by recording a game where two players were to create an army and defeat the opponent. The players could either be two computer controlled (bots), or a bot and a human player. The participants observing the replays were asked if they thought each player was controlled by a bot or a human, and motivate their answer. The goal of the experiment was to see if we could find some characteristics of what makes a RTS player more humanlike. Some might draw a parallel with the famous Turing test, but our aim is not to develop a bot that plays like a human would, but rather try to find what is perceived as human like decisions in an RTS game.

The same experiment setup was used by Livingstone in (Livingstone, 2006) but with the well known game Pong instead of an RTS game as we use. Livingstone concludes that this experiment setup cannot be called a Turing Test since, for example, there is no interaction between the participants and the AI bot. He also concludes that believability tests can provide interesting insights and ideas for improvements and can be very useful if care is taken when setting up the experiments and analyzing the results.

10.1.3 Outline

First, we will describe the environment and how we generated the replays that were used in the study. This is followed by a description of the experimental setup. In Section 10.4 we describe the results and a discussion about them are presented in Section 10.5. We finish by presenting some ideas about future work.

10.2 Generating replays

In the experiment we used the open-source RTS game engine Spring and the mod Evolution RTS (Spring Project, 2009). Evolution RTS is a science fiction mod where the players have access to a wide range of different futuristic tanks and buildings. In the experiments only ground units were available. Players were not allowed to build any aircraft- or hovering units. The ground units range from fast, light armored scout tanks to large, very heavy but slow tanks with massive firepower. There are also two different artillery units which can deal heavy damage but have problems hitting moving targets. To defend a base two stationary defensive towers, one light and one heavy, are available. The players also need to gather resources in form of energy (gained from solar collectors and power plants) and metal (gathered from metal extractors placed at specific areas on the map).

Five persons were asked to play one game each against a bot. The persons were given a brief introduction to the game and played a practice game against a bot available in the Spring repository (RAI version 0.601) before the actual logging took place. The recordings started once the second game began and were stopped when one of the players were defeated. The five persons participating in the recordings had varying experience in RTS games.

The bots were implemented using the multi-agent potential field approach described Papers I, III, V and VI. Two different bots were created, called Largebase and Tankrush. The details of each bot are described in Sections 10.2.1 and 10.2.2.

The two bots come in three different versions. The differences are in the paces in which the bots perform their actions. In version 1 the bot immediately executes an action once it has decided what to do. In version 2 there is a slight delay between when an action is finished and when a new one can be started. Version 3 has a slightly higher delay than version 2. The delay slows down the bot and can be perceived as having a more humanlike thinking and reaction time. The delay is random between certain intervals. Version 2 has a delay between 4 and 8 seconds, and version 3 a delay between 8 and 15 seconds, see Table 10.1.

Fourteen replay games were recorded for the experiment (see Table 10.2). In five of those a human was playing, and in the other nine different versions of the bots were playing against each other.

10.2.1 Largebase

The Largebase bot focuses on building a large base and using heavy units. First it creates ten light scout tanks which he uses for an early surprise attack. If the opponent is still alive the bot builds a second base in an opposite corner of the map. Both bases are

Table 10.1: *The six bots used in the experiment. The delay is the number of seconds it waits after executing one action, before it starts executing the next.*

	delay (in s)		
Bot	0	4–8	8–15
Largebase	L_1	L_2	L_3
Tankrush	T_1	T_2	T_3

protected by 2-4 light defensive towers. The bot builds groups of the most heavy tank available (Fatso) with some support from artillery units. Once a group of four heavy tanks and 2 artillery units has been built, the group locates and attacks the enemy. The same group setup is used in the rest of the game.

10.2.2 Tankrush

The Tankrush bot focuses on quickly getting some resource income and then build groups of medium heavy tanks supported by artillery. Once a group of 5 tanks and 3 artillery units have been built, the group is sent to attack the enemy. The same tactic is used throughout the game. The bot does not use any defensive towers.

10.2.3 The human players

Four different human players participated in the replay games; we call them Mike, Pete, George and Simon. The former three are PhD students, and the latter one is a student at the bachelor level.

Mike as well as *George* played RTS games around 10 years ago and they consider themselves to be averagely skilled RTS players. *Simon* is an experienced gamer, but he has not played very much RTS games. *Pete* is a quite experienced player with an understanding of RTS game AI mechanics.

10.3 Experimental Setup

The experiments were carried out at DreamHack Winter 2009, the largest LAN party in the world. We were positioned at the booth of our University, where players who stopped at our booth were asked if they would like to participate in a scientific experiment. Those who agreed were told that they should watch a replay of a RTS game, and what they were to observe and make a decision about. During the game the participants were informed about the type of units and structures each player was using, and the strengths

Table 10.2: *The fourteen replays used in the experiment.*

Match	Player 1	Player 2	Winner	Time (min)
M_1	L_1	R_1	L_1	12
M_2	L_1	R_2	L_1	13
M_3	L_1	R_1	L_1	19
M_4	L_2	$Mike$	$Mike$	12
M_5	L_3	R_1	L_3	15
M_6	L_2	R_2	L_2	12
M_7	L_1	$Pete$	$Pete$	28
M_8	L_3	R_3	L_3	14
M_9	L_1	R_3	L_1	13
M_{10}	L_2	$George$	$George$	18
M_{11}	L_3	R_2	L_3	24
M_{12}	R_2	$Simon$	$Simon$	14
M_{13}	L_2	R_3	L_2	11
M_{14}	L_1	$Mike$	$Mike$	32

and shortcomings of the units. Once the participant has made up a decision of if the players were human- or bot-controlled, they were asked to fill in a questionnaire.

The same experiment was carried out at a local LAN party held at a high school close to our University.

10.3.1 The Questionnaire

The Questionnaire consisted of a brief instruction on how to fill in the answers, a number of general questions (concerning age, sex and experience of playing RTS games), and a section for each of the two players. Each section contains three word pairs where the participants were to rank how they perceived the player played the game, a checkbox asking if the participant thought the player were a bot or a human, and a free text motivation for their decision (see Fig. 10.1). The word pairs used are listed in Table 10.3. For each word pair the participants gave a score between 1 (the player is best described by word 1) and 7 (the player is best described by word 2). A score of 4 means that the participant does not think any of the two words describes the player.

Table 10.3: *A list of the word pairs used in the questionnaire.*

Word 1	Word 2
Predictable	Varied
Easily beaten	Demanding
Boring	Entertaining

10.4 The Results

In total 56 persons participated in the experiment. A majority of the participants were between age 16 and 20. Of the participants 52 were male and 4 female.

The results will be presented in detail for each bot individually. The upper part of Table 10.4–10.9 describes in what match (M_1–M_{14}) the bot played, what opponent it faced, and the quotient between how many of the participants that judged it to be a human player (H) and a computer bot (C). The last three columns refer to the judgement of the predictability (Pred.), the strength and the level of entertainment (E-mnt) it would provide. Each of these values is an average of the participants' scores along an ordinal scale 1–7.

The lower part of the tables present the written arguments that the participants left to defend their guesses of which players were human and which where bots. These comments have been translated from Swedish. Each separate quote is the translation of the answer of one participant, but not all participants commented on their guesses (so there may be more participants that studied a match than the ones commenting on their classification of the players).

In Table 10.10 we show the results of the human players in a way similar to the one for the bots.

10.4.1 The Largebase Bots

The results of the Largebase bots L_1–L_3 is shown in Tables 10.4–10.6.

10.4.2 The Tankrush Bots

In Tables 10.7–10.9 the participants opinions about the Tankrush bots R_1–R_3 are shown.

10.4.3 Human player results

Table 10.10 shows the results of the participants' opinions about the human players.

Table 10.4: *Results of the matches with the L_1 bot.*

Match	Opponent	H/C?	Pred.	Strength	E-mnt
M_1	R_1	3/2	4.4	5.8	5.6
M_2	R_2	3/0	5.7	4.0	4.0
M_7	$Pete$	2/1	4.3	4.3	4.0
M_9	R_3	1/2	5.3	4.3	5
M_{14}	$Mike$	1/1	4.0	2.0	4.0
Average		10/6	4.75	4.44	4.69
Hum.?	"He is thinking strategically and awaits the opponent.", "More aggressive, built. Began light and changed to heavier. Moved its units.", "Just a feeling.", "He is too flexible to be a bot.", "Uses self destruct and scouts with small units. Builds good counters against opponent.", "More active. Spread out fast. Fast attack. Concentrated attacks.", "He changed his tactics very often", "Plays more defensive.", "He moves straight on."				
Comp.?	"It moves all units separately.", "Fast to perform new tasks once a building is finished.", "Did not attack with its full strength.", "Feels as if he controlled most of his units simultaneously, which made it feel like a bot. Also went straight towards the opponent base."				

10.4.4 Human-like vs. bot-like RTS game play

In Figures 10.2–10.4 the participants' expressed opinions about the predictability, the strength and the entertainment value of the bots, as well as the players, are shown. Although there seem to be no general conclusions that can be drawn based upon the results, there seems to be a correlation between the predictability of a player, and the strength of it.

The next step is to categorize the comments into different groups. This can of course be done in several ways. We have identified the following: *simultaneous movements, overall strategy, attack tactics, placement of buildings, level of activity, ability to adapt, build order, player strength, map knowledge, intelligence and creativity,* and *monotony.* There is also a number of very vaguely expressed *feelings* that it is human or computer based. These comments are collected in an *emotional judgement* class. If a participant left a motivation spanning over more than one category, we have filed it as several

Table 10.5: *Results of the matches with the L_2 bot.*

Match	Opponent	H/C?	Pred.	Strength	E-mnt
M_3	R_1	6/2	4.1	3.4	3.4
M_4	$Mike$	3/2	5.4	3.2	4.0
M_6	R_2	0/3	4.3	4.3	3.7
M_{10}	$George$	0/6	3.8	3.3	3.3
M_{13}	R_3	1/3	3.2	3.5	4
Average		10/16	4.19	3.46	3.62
Hum.?	"Looks like it, the way he plays.", "It feels as if there was a thought behind each move, even though it did not succeed fully.", "In the way he built and moved.", "He/it does not have the time to move its units, since humans cannot do that much simultaneously in strategy games.", "The way the buildings were placed", "Seemed to make human mistakes that a computer never would have done.", "Red is poor at doing lot of things simultaneously.", "Built so little."				
Comp.?	"I personally think it had a quite CPU-like pattern in its game play.", "Calm play. Hazardous unit build planning. Too little 'base' defense line.", "Consequent strategy", "He sent bad units on a good unit to defend a metal extractor. Not very large advancement.", "Even a beginner would have been better.", "It is fast. I would have guessed at high APM.", "Faster, kind of.", "It builds so much"				

comments (which is why the sum of experiments does not fully match the number of comments).

Simultaneous movements

The ability to control many units individually and simultaneously is a typical indicator of a computer player, according to the participants. 9 out of 9 comments also correctly identified bots using this property, compared to 1 out of 3 correct identifications of human players. The two that did not manage to be helped by this, guessed that it was human players because they moved slowly or seemed unable to make things simultaneously.

Table 10.6: *Results of the matches with the L_3 bot.*

Match	Opponent	H/C?	Pred.	Strength	E-mnt
M_5	R_1	3/0	6	4.3	3
M_8	R_3	2/1	4.7	6	6
M_{11}	R_2	2/3	4.4	5.2	3.8
Average		7/4	4.91	5.18	4.18
Hum.?	"He was unpredictable, not like a bot.", "He used a lot of dynamic methods that feels more typical to an experienced player than a bot.", "It builds another camp", "Two bases", "He seems to know where the opponent is, even though he has not discovered that much of the map.", ""				
Comp.?	"He did not build in patterns as 2 did. And he controlled several men simultaneously, he built strong defense.", "Builds small, unimproved tanks. Attacks unsmart.", "He was the more eager one and attacked almost immediately.",				

Overall strategy

This includes general strategic impressions such as the willingness to attack and to defend itself. In total 17 comments were on the general strategy of which 10 were in favor of a human player, and 7 of them supported a bot. 6 guesses of each sort were correct.

Placement of buildings

This category includes comments about where the players placed their buildings, often in relation to the buildings already built.

Build planning

Although seemingly similar to the previous category, this covers the order in which the players seemed to build their units (both buildings and mobile units such as tanks). From the figures we can see that it is seems to be a far better indicator for identifying bots, than to identify human players.

Tactics

The tactics was one of the most successful indicators of whether the player was human or not, and that applies to both types of players. The category consisted mainly of

Table 10.7: *Results of the matches with the R_1 bot.*

Match	Opponent	H/C?	Pred.	Strength	E-mnt
M_1	L_1	2/3	3.2	3.8	3.6
M_3	L_2	2/6	4.4	4.9	4.3
M_5	L_3	0/3	3.7	4	4.3
Average		4/12	3.88	4.38	4.06
Hum.?	"Organized attacks.", "He runs more ’tactics’. He tries to take one base at a time."				
Comp.?	"He goes ’straight for it’", "Does not only produce the same tanks. Not very defensive.", "He was predictable.", "Same as 1 (I personally think it had a quite CPU-like pattern in its game play.)", "Stayed and patrolled unnecessary areas.", "All units moved at once.", "The structure of the base and the way the machines moved.", "Always has its troops in movement.", "Use a lot of men at the same time.",				

comments on the way it attacked the opponent.

Level of activity

If it seemed slow, or fast, the comment on it ended up here. This was in our experiment, a reasonably good indicator of a computer player.

Player strength

Not very many participants commented on the strength of the player, but for those who did, the guesses were correct.

Game world knowledge

Behind this indicator is both the ability to seem to know where the opponent is (i.e. a perceived lack of fog of war) as well as the (lack of) ability to perform exploratory tasks. Also this seems to be quite an informative indicator when it comes to make qualified guesses.

Table 10.8: *Results of the matches with the R_2 bot.*

Match	Opponent	H/C?	Pred.	Strength	E-mnt
M_2	L_1	0/3	3.3	1.7	2.7
M_6	L_2	2/1	3.0	1.7	2.3
M_{11}	L_3	3/2	3.2	3.2	2.6
M_{12}	$Simon$	2/1	3.7	3.7	4.0
Average		7/7	3.29	2.64	2.86
Hum.?	"He did not do very much.", "He built in patterns and without defense towers.", "He built crosswise.", "It upgraded first before it attacked.", "Same thing, they kind of know where the other is from the start. They do not explore and starts going with fairly strenghts at once."				
Comp.?	"It attacks the most.", "Very still. Did not move its units. Did not change tactics.", "Too correct built.", "Makes it easy for the opponent.", "Monotonous.", "A slightly better bot that upgrades, but it did not look like a human player."				

Intelligence and creativity

This category includes comments about the level of intelligence of the player, e.g. referring to "human mistakes that a computer never would make". This indicator was not very successful for the participants and most of the guesses failed.

Ability to adapt

The ability to adapt was according to the participants an ability that human players had. All of them were in this case wrong in their guesses.

Monotony

Monotony refers to the identification of repetitive actions (or lack thereof).

Emotional judgement

In this category, we included vaguely expressed statements such as: "CPU-like pattern in its play" (bot), or "Looks like it the way he plays" (human).

Table 10.9: *Results of the matches with the R_3 bot.*

Match	Opponent	H/C?	Pred.	Strength	E-mnt
M_8	L_3	1/2	2.0	1.7	2.3
M_9	L_1	1/2	3.3	3.7	3.0
M_{13}	L_2	3/1	3.5	3.8	2.8
Average		5/5	3.00	3.10	2.70
Hum.?	"Felt a bit new to the game. Did not go straight for the opponent. Did not retire when he should.", "Classical opening moves. Focus on collecting resources.", "It plays somewhat slower, but stronger.", "More careful and safe?"				
Comp.?	"It moves all units separately.", "He played generally stiff, sent units at an overwhelming defense without retiring and expanded very little.", "Same as 1 (in the way he built and moved)", "It is very slow and dull.", "It built in a middle of a war.",				

No comment

A quite substantial part of the participants did not leave any (or just a partial) motivation to their guesses. In all, 33 guesses were left without comments. 21 of them were correct, and 12 incorrect.

10.5 Discussion

The aim of the study was to see if we could find some characteristics of human like behavior in RTS games. The aim was not to create a bot that behaved like a human, neither to show that human players were most often perceived as humans in the replays (even though we provided these numbers in the paper).

The results, when categorized, showed some interesting things. *The movement of the units* is one thing that differs between bots and human players. Bots are able to simultaneously move different units in different directions which is not possible for the average human player. This could be prevented if bots grouped units in squads and let all units in a squad move in the same direction, and not let several squads issue new move orders at the same time.

Properties like *intelligence, creativity* and *adaptivity* were however not a clear way to differ between bots and humans. Although the majority of participants thought these characteristics were humanlike, some thought that it instead was a sign of a well de-

Table 10.10: *Results for the human players.*

Match	Player	Opp.	H/C?	Pred.	Str.	E-mnt
M_4	$Mike$	L_2	2/3	3.2	4.8	5.0
M_7	$Pete$	L_1	1/2	4.0	3.7	2.0
M_{10}	$George$	L_2	6/0	5.5	5.2	5.7
M_{12}	$Simon$	R_2	1/2	5.0	6.0	4.3
M_{14}	$Mike$	L_1	2/0	3.5	4.0	4.5

$Mike$	Hum.?	A bit monotonous. Goes directly for the largest tank and thinks little about defense and the tactics of the opponent.", "The yellow attacked directly", "It is slow and do not know what it is doing."
$Mike$	Comp.?	"Did exactly what you could expect. No real surprises.", "Built too much."
$Pete$	Hum.?	"Forgot units as he was being attacked."
$Pete$	Comp.?	"A little bit too passive.", "He was very defensive and did the same thing all the time."
$George$	Hum.?	"More impulse.", "The way the units were moved."
$Simon$	Hum.?	"Thinks more about defense."

signed bot. Some people thought a bot shall react to the opponent and try different unit setups, and that most bots are implemented in that way.

Level of activity and *player strength* were reasonably good indicators. A player that does too many things will be perceived as a bot, while players with low activity are often seen as inexperienced humans. A player that wins his or her match is often seen as a good human player and rarely a bot. Note that we did not allow the bots to cheat in any way (e.g. by removing fog of war or let their units get more hit points than the human player's units).

General strategy and *build planning* were not very good indicators. A player that did something clever was as often perceived as a well implemented bot as an experienced human player. The same goes for bad tactics; it could be a sign of a bad bot or an indication of an inexperienced human. *Tactics* of attacking or defending units and physical placement of buildings were better indicators, but there were few comments in these categories.

Knowledge about the game world also seems to be a pretty good indicator. Bots were more keen to go directly to the most logical spot for the opponent player to be located

Table 10.11: *Classification accuracy of the different types of comments.*

		Correct guesses		Incorr. guesses	
Comment category	Total	Human	Bot	Human	Bot
Simult. movements	12	1	9	2	0
General Strategy	17	6	6	4	1
Placement of buildings	4	2	2	0	0
Build planning	17	4	8	4	1
Tactics	9	4	4	1	0
Level of activity	10	2	6	1	1
Player strength	4	2	2	0	0
Game world knowledge	7	3	3	1	0
Intelligence/creativity	6	2	0	3	1
Ability to adapt	3	0	0	3	0
Monotony	7	1	4	1	1
Emotional judgement	7	2	3	2	0
Sum of comments	103	29	47	22	5
No comment given	33	5	16	9	3

(i.e. the opposite of the map) while players did more scouting. This was noticed and commented by some of the participants.

10.5.1 On the method

If we look back; what could we have improved method wise? Although this series of experiments as such is closed, we would like to share some thoughts on this.

First, the variation of the matches could be larger. We used two bots in three different versions, but none of them were tuned to be strong enough to beat any of the human participants. Given that, we cannot say that much about how a strong bot would be perceived by the participants of the experiment. We also restricted the bots to the type of bots that we have implemented ourselves. Maybe a more traditional path-finding-based bot would have been easier to detect for the participants?

Second, due to a random pick of matches (the participants were given one of the matches at random), there were quite large differences between the most frequently watched game, and the least. This in turn makes it harder to analyze the results and draw general conclusions about the different bots.

Third, we should also have had matches with two human players. For technical reasons, this turned out to be hard to achieve with the setup we used, but we think that it may contribute to the results to also have two human players. A majority of the participants guessed at one human and one bot (even when it was two bots), so for the sake of completeness, we should also have had two human players.

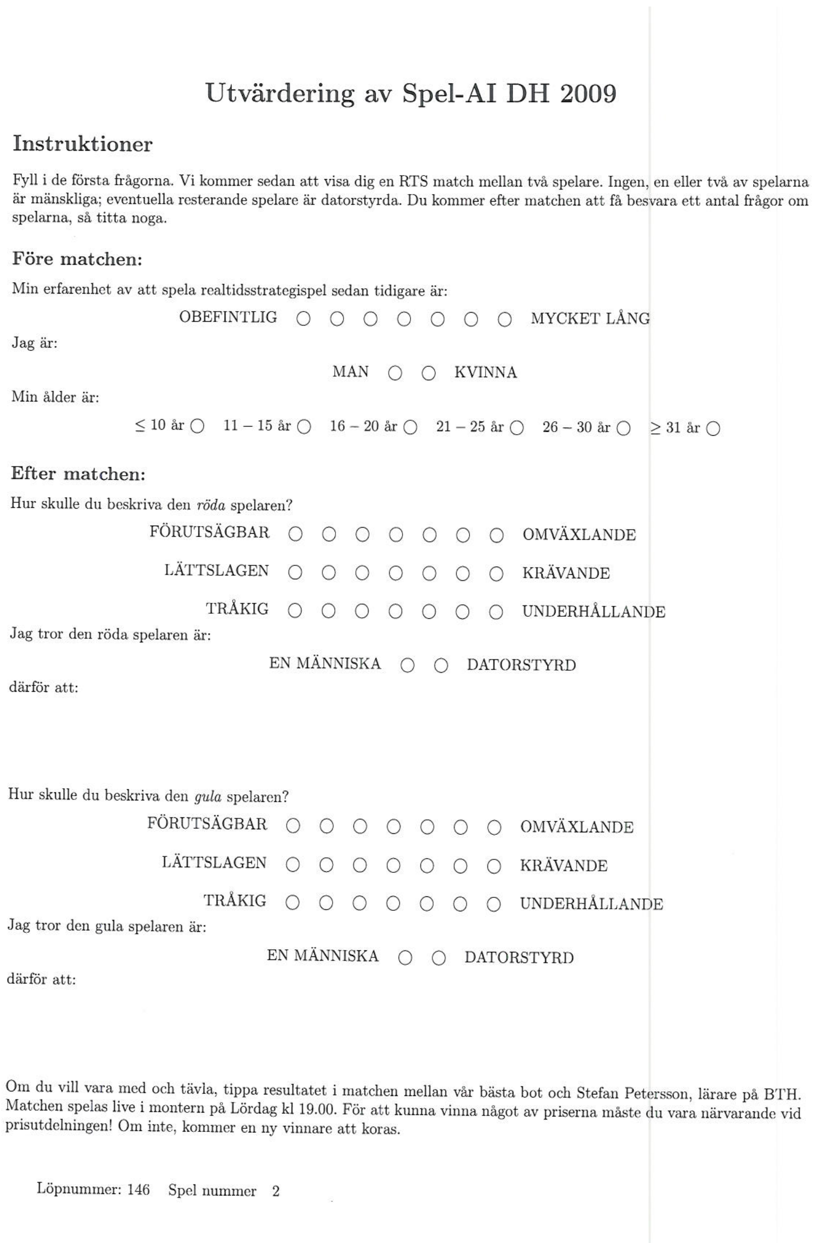
10.6 Conclusions

There are a few indicators that we can conclude are associated with human like RTS game play. The micro-management of units, the placement of buildings, tactics, player strength, and game world knowledge are all indicators that the participants used successfully to argue for a correct guess. Intelligence, creativity and ability to adapt are however less successful signs (in our experiment).

We must also point out that the number of participants in our experiments was relatively small and a larger sample would give us more knowledge about the categories with less strong indications.

10.7 Future work

We believe our findings can be used to design future experiments that can give us even better insight in human characteristics in RTS games. We can learn from our results and create adaptive bots that are not allowed to simultaneously move units in different directions and that does not cheat regarding game world knowledge. If the same or similar experiments would be conducted with the re-designed bots it could give us even more insight in human- versus bot-like play for RTS games. A good insight in human gameplay can be important when creating good AI players. As Yannakakis and Hallam points out, humanlike bots does not always have to be fun to play against, but we believe it is a good start to be aware of humanlike characteristics when designing them (Yannakakis & Hallam, 2007). Another issue worth further investigations is to find out how and why there seems to be a connection between the predictability of a player, and the strength of it.



Utvärdering av Spel-AI DH 2009

Instruktioner

Fyll i de första frågorna. Vi kommer sedan att visa dig en RTS match mellan två spelare. Ingen, en eller två av spelarna är mänskliga; eventuella resterande spelare är datorstyrda. Du kommer efter matchen att få besvara ett antal frågor om spelarna, så titta noga.

Före matchen:

Min erfarenhet av att spela realtidsstrategispel sedan tidigare är:

OBEFINTLIG ○ ○ ○ ○ ○ ○ ○ MYCKET LÅNG

Jag är:

MAN ○ ○ KVINNA

Min ålder är:

$\leq$ 10 år ○ 11 – 15 år ○ 16 – 20 år ○ 21 – 25 år ○ 26 – 30 år ○ $\geq$ 31 år ○

Efter matchen:

Hur skulle du beskriva den *röda* spelaren?

FÖRUTSÄGBAR ○ ○ ○ ○ ○ ○ ○ OMVÄXLANDE

LÄTTSLAGEN ○ ○ ○ ○ ○ ○ ○ KRÄVANDE

TRÅKIG ○ ○ ○ ○ ○ ○ ○ UNDERHÅLLANDE

Jag tror den röda spelaren är:

EN MÄNNISKA ○ ○ DATORSTYRD

därför att:

Hur skulle du beskriva den *gula* spelaren?

FÖRUTSÄGBAR ○ ○ ○ ○ ○ ○ ○ OMVÄXLANDE

LÄTTSLAGEN ○ ○ ○ ○ ○ ○ ○ KRÄVANDE

TRÅKIG ○ ○ ○ ○ ○ ○ ○ UNDERHÅLLANDE

Jag tror den gula spelaren är:

EN MÄNNISKA ○ ○ DATORSTYRD

därför att:

Om du vill vara med och tävla, tippa resultatet i matchen mellan vår bästa bot och Stefan Petersson, lärare på BTH. Matchen spelas live i montern på Lördag kl 19.00. För att kunna vinna något av priserna måste du vara närvarande vid prisutdelningen! Om inte, kommer en ny vinnare att koras.

Löpnummer: 146 Spel nummer 2

Figure 10.1: *The questionnaire used in the experiments.*

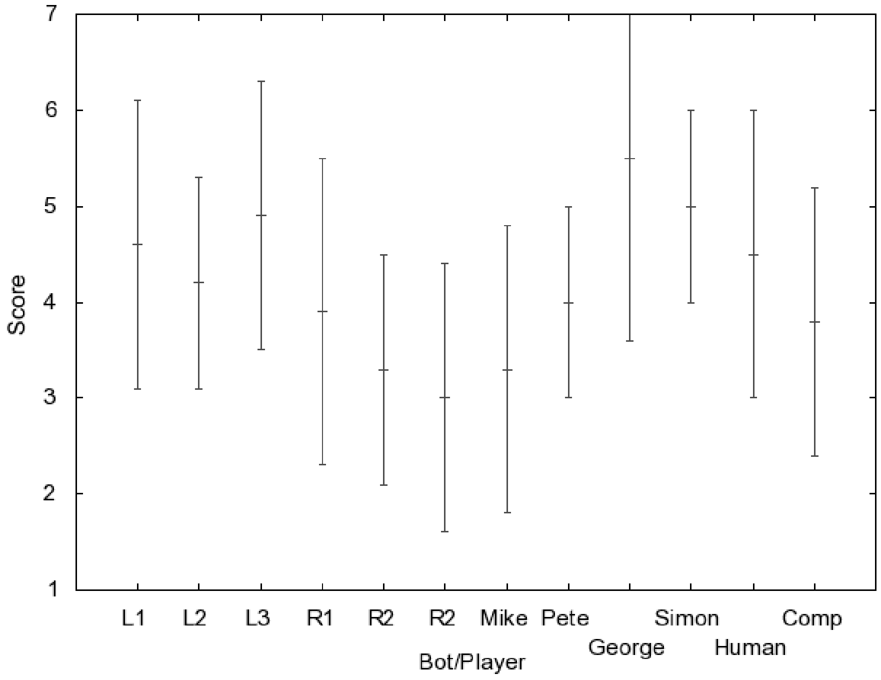


Figure 10.2: *The predictability of the players according to the participants.*

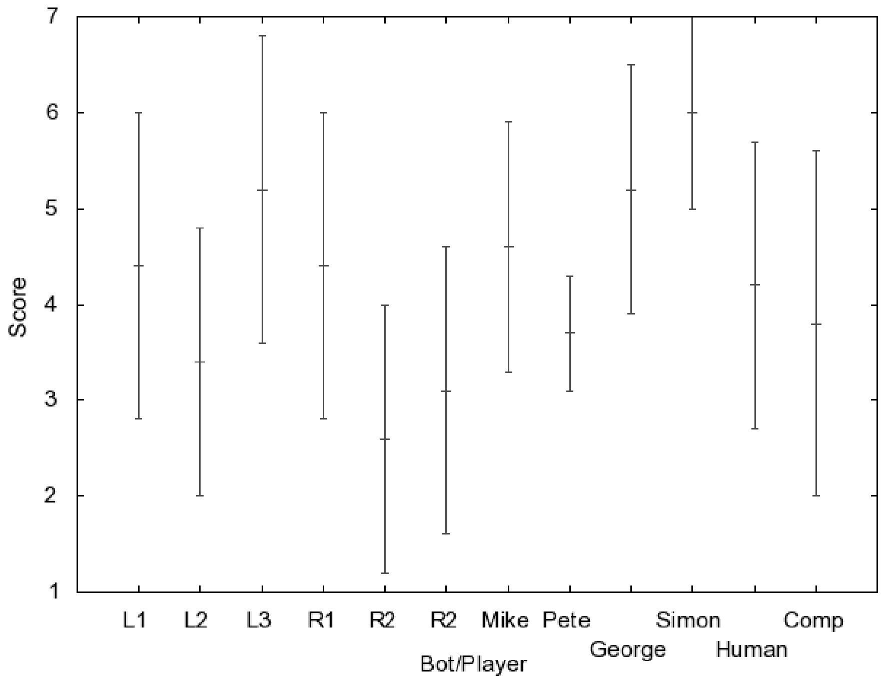


Figure 10.3: *The strength of the players according to the participants.*

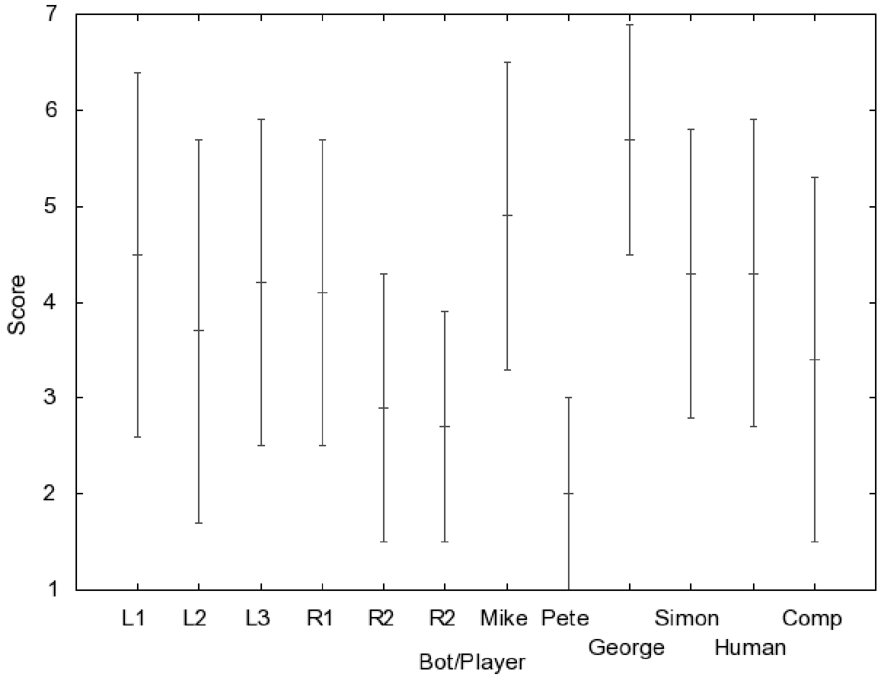


Figure 10.4: *The entertainment value of the players according to the participants.*

REFERENCES

AIIDE 2010 StarCraft Competition. (2010). Retrieved 2011-12-07, from `eis.ucsc.edu/Tournament4Results`

Alexander, B. (2006). Flow Fields for Movement and Obstacle Avoidance. In *AI Game Programming Wisdom 3.* Charles River Media.

Arkin, R. C. (1987). Motor schema based navigation for a mobile robot. In *Proceedings of the IEEE International Conference on Robotics and Automation.*

Bakkes, S., Spronck, P., & Herik, J. van den. (2008). Rapid Adaptation of Video Game AI. In *Proceedings of 2008 IEEE Symposium on Computational Intelligence and Games (CIG).*

Borenstein, J., & Koren, Y. (1989). Real-time obstacle avoidance for fast mobile robots. *IEEE Transactions on Systems, Man, and Cybernetics.*

Borenstein, J., & Koren, Y. (1991). The vector field histogram: fast obstacle avoidance for mobile robots. *IEEE Journal of Robotics and Automation.*

Brooks, R. (1986). A robust layered control system for a mobile robot. In *IEEE Journal of Robotics and Automation* (p. RA-2(1):1423).

Buckland, M. (2005). *Programming Game AI by Example.* Wordware Publishing, Inc.

Buro, M. (2007a). *ORTS — A Free Software RTS Game Engine.* Retrieved 2011-12-07, from `www.cs.ualberta.ca/~mburo/orts/`

Buro, M. (2007b). *ORTS RTS game AI competition, 2007.* Retrieved 2011-12-07, from `www.cs.ualberta.ca/~mburo/orts/AIIDE07`

Buro, M. (2008). *ORTS RTS game AI competition, 2008.* Retrieved 2011-12-07, from `www.cs.ualberta.ca/~mburo/orts/AIIDE08`

Buro, M., & Furtak, T. (2004). RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS).*

BWAPI - An API for interacting with StarCraft: Broodwar. (2009). Retrieved 2011-12-07, from `code.google.com/p/bwapi`

Cerpa, D. H. (2008). A Goal Stack-Based Architecture for RTS AI. In *AI Game Programming Wisdom 4.* Charles River Media.

deJong, E. (2004). Intransitivity in coevolution. In Parallel Problem Solving from Nature - PPSN VIII. In *volume 3242 of Lecture Notes in Computer Science.* Springer.

Demyen, D., & Buro, M. (2008). Fast Pathfinding Based on Triangulation Abstractions. In *AI Game Programming Wisdom 4.* Charles River Media.

Dill, K. (2006). Prioritizing Actions in a Goal-Based RTS AI. In *AI Game Programming Wisdom 3.* Charles River Media.

DreamHack. (2009). Retrieved 2011-12-07, from `www.dreamhack.se`

Electronic Sports. (2011). Retrieved 2011-12-07, from `http://en.wikipedia.org/wiki/Electronic_sports`

Freed, M., Bear, T., Goldman, H., Hyatt, G., Reber, P., & Tauber, J. (2007). Towards more human-like computer opponents. *AAAI Technical Report SS-00-02.*.

Game Balance. (2011). Retrieved 2011-12-07, from `en.wikipedia.org/wiki/Game_balance`

Game Sales Charts. (2011). Retrieved 2011-12-07, from `www.pvcmuseum.com/games/charts/computer-and-video-game-market-sales.htm`

Hart, P. E., Nilsson, N. J., & Raphael, B. (1972). Correction to A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *SIGART Newsletter*(37), 28-29.

Higgins, D. (2002). How to Achieve Lightning-Fast A*. In *AI Game Programming Wisdom.* Charles River Media.

Howard, A., Matarić, M., & Sukhatme, G. (2002). Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem. In *Proceedings of the 6th International Symposium on Distributed Autonomous Robotics Systems (DARS02).*

Johansson, S. J. (2006). On using multi-agent systems in playing board games. In *Proceedings of Autonomous Agents and Multi-agent Systems (AAMAS).*

Johansson, S. J., & Saffiotti, A. (2002). An electric field approach to autonomous robot control. In *RoboCup 2001.* Springer Verlag.

Johnson, G. (2006). Avoiding Dynamic Obstacles and Hazards. In *AI Game Programming Wisdom 2.* Charles River Media.

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*.

Khatib, O. (2004). Human-like motion from physiologically-based potential energies. In J. Lenarcic & C. Galletti (Eds.), *On Advances in Robot Kinematics* (pp. 149–163). Kluwer Academic Publishers.

Koenig, S. (2004). A comparison of fast search real-time situated agents. In *Proceedings of Autonomous Agents and Multi-agent Systems (AAMAS).*

Koenig, S., & Likhachev, M. (2006). Real-Time Adaptive A*. In *Proceedings of Autonomous Agents and Multi-agent Systems (AAMAS).*

Kraus, S., & Lehmann, D. (1995). Designing and building a negotiating automated agent. *Computational Intelligence*, 11(1):132171.

Livingstone, D. (2006). Turings Test and Believable AI in Games. *Computers in*

Entertainment (CIE).

Macaloon, J. J. (2007). *J. J. Macaloon, This Great Symbol: Pierre de Coubertin and the Origins of the Modern Olympic Games, 1st ed. Routledge, November 2007. [Online].* Retrieved 2011-12-07, from `www.amazon.ca/gp/product/041549494X`

Malone, T. W. (1981a). Toward a theory of intrinsically motivating instruction. *Cognitive Science*(4), 333–370.

Malone, T. W. (1981b). What makes computer games fun? *BYTE*(6), 258–277.

Mamei, M., & Zambonelli, F. (2004). Motion coordination in the Quake 3 Arena environment: A field-based approach. In *Proceedings of Autonomous Agents and Multi-Agent Systems (aamas).* IEEE Computer Society.

Massari, M., Giardini, G., & Bernelli-Zazzera, F. (2004). Autonomous navigation system for planetary exploration rover based on artificial potential fields. In *Proceedings of Dynamics and Control of Systems and Structures in Space (DCSSS) 6th Conference.*

Nareyek, A. (2004). AI in computer games. In *ACM Queue 1(10)* (p. 58-65).

Niederberger, C., & Gross, M. H. (2003). *Towards a Game Agent* (Tech. Rep.). CS Technical Report #377.

Olesen, J. K., Yannakakis, G. N., & Hallam, J. (2008). Real-time challenge balance in an RTS game using rtNEAT. In *Proceedings of 2008 IEEE Symposium on Computational Intelligence and Games (CIG).*

Olsson, P.-M. (2008). Practical Pathfinding in Dynamic Environments. In *AI Game Programming Wisdom 4.* Charles River Media.

Pittman, D. (2008). Command Hierarchives Using Goal-Oriented Action Planning. In *AI Game Programming Wisdom 4.* Charles River Media.

Reynolds, J. (2002). Tactical Team AI Using a Command Hierarchy. In *AI Game Programming Wisdom.* Charles River Media.

Röfer, T., et al. (2004). *GermanTeam 2004 - the German National RoboCup team.*

Schwab, B. (2009). AI Game Engine Programming. In *AI Game Engine Programming.* Charles River Media.

Scott, B. (2002). The Illusion of Intelligence. In *AI Game Programming Wisdom.* Charles River Media.

Silver, D. (2006). Cooperative Pathfinding. In *AI Game Programming Wisdom 3.* Charles River Media.

Soni, B., & Hingston, P. (2008). Bots trained to play like a human are more fun. In *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN 2008).*

Spring Project. (2009). Retrieved 2011-12-07, from `springrts.com`

StarCraft 2 sales blitz a success. (2010). Retrieved 2011-12-07, from `arstechnica.com/gaming/news/2010/08/starcraft-2-sales-blitz-a-success-sales-crush-console-games.ars`

Sun, X., Koenig, S., & Yeoh, W. (2008). Generalized Adaptive A*. In *Proceedings of Autonomous Agents and Multi-agent Systems (AAMAS).*

Sweetster, P., & Wyeth, P. (2005). Gameflow: A model for evaluating player enjoyment in games. *ACM Computers in Entertainment*, *3*(3).

Synnaeve, G., & Bessiere, P. (2011). A Bayesian Model for Opening Prediction in RTS Games with Application to StarCraft. In *Proceedings of 2011 IEEE Symposium on Computational Intelligence and Games (CIG).*

Table of Counters - StarCraft Atrium. (2011). Retrieved 2011-12-07, from `archive.rpgclassics.com/subsites/starcraft/counters.shtml`

Thurau, C., Bauckhage, C., & Sagerer, G. (2004a). Imitation learning at all levels of game-AI. In *Proceedings of the International Conference on Computer Games, Artificial Intelligence, Design and Education.* University of Wolverhampton.

Thurau, C., Bauckhage, C., & Sagerer, G. (2004b). Learning human-like movement behavior for computer games. In *Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior (SAB'04).*

Tomlinson, S. L. (2004). The long and short of steering in computer games. *International Journal of Simulation Systems, Science and Technology*.

Tozour, P. (2004). Using a Spatial Database for Runtime Spatial Analysis. In *AI Game Programming Wisdom 2.* Charles River Media.

Vadakkepat, P., Tan, K. C., & Ming-Liang, W. (2000). Evolutionary artificial potential fields and their application in real time robot path planning. In *Proceedings of the 2000 Congress on Evolutionary Computation* (pp. 256–263). IEEE Press.

vanLent, M., Laird, J., Buckman, J., Hartford, J., Houchard, S., Steinkraus, K., et al. (1999). Intelligent agents in computer games. In *Proceedings of AAAI.*

Warcraft III Game reviews. (2008). Retrieved 2011-12-07, from `reviews.cnet.com/pc-games/warcraft-iii-the-frozen/4505-9696_7-30602384.html`

Warcraft III Guide. (2008). Retrieved 2011-12-07, from `guides.ign.com/guides/12906/page_6.html`

Weber, B., & Matteas, M. (2009). A Data Mining Approach to Strategy Prediction. In *Proceedings of 2009 IEEE Symposium on Computational Intelligence and Games (CIG).*

Wirth, N., & Gallagher, M. (2008). An Influence Map Model for Playing Ms. Pac-Man. In *Proceedings of 2008 IEEE Symposium on Computational Intelligence and Games (CIG).*

Yannakakis, G. N., & Hallam, J. (2007). Towards Optimizing Entertainment in Computer Games. *Applied Artificial Intelligence*, 21:933-971.