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

Johan Hagelbäck and Stefan J. Johansson

Abstract— Bots for Real Time Strategy (RTS) games provide a rich challenge to implement. A bot controls a number of units that may have to navigate in a partially unknown environment, while at the same time search for enemies and coordinate attacks to fight them down. It is often the case that RTS AIs cheat in the sense that they get perfect information about the game world to improve the performance of the tactics and planning behavior. We show how a multi-agent potential field based bot can be modified to play an RTS game without cheating, i.e. with incomplete information, and still be able to perform well without spending more resources than its cheating version in a tournament.

I. 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, *"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 [1].

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.

Johan Hagelbäck and Stefan J. Johansson are with the Department of Software and Systems Engineering, Blekinge Institute of Technology, Box 520, SE-372 25, Ronneby, Sweden. e-mail: `jhg@bth.se, sja@bth.se`.

A. 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.

B. 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.

II. ORTS

Open Real Time Strategy (ORTS) [2] 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 [2].

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 [3].

A. Descriptions of Opponents

The following opponents were used in the experiments:

1) NUS: The team *NUS* uses finite state machines and influence maps in high-order planning on group level. The units in a group spread out on a line and surround the opponent units at *Maximum Shooting Distance* (MSD). Units use the cool-down period to keep out of MSD. Pathfinding and a flocking algorithm are used to avoid collisions.

2) UBC: This team gathers units in squads of 10 tanks. Squads can be merged with other squads or split into two during the game. Pathfinding is combined with force fields to avoid obstacles and a 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.

3) WarsawB: The *WarsawB* team uses pathfinding with an additional dynamic graph for moving objects. The units use 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 attack the weakest opponent unit in range.

4) Uofa06: Unfortunately, we have no description of how this bot works, more than that it was the winner of the 2006 year ORTS competition. Since we failed in getting the 2007 version of the UofA bot to run without stability problems under the latest update of the ORTS environment, we omitted it from our experiments.

III. MULTI-AGENT POTENTIAL FIELDS

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 [4]. Later on Arkin [5] 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. [6], [7]. 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 [8].

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. developed a mobile sensor network deployment using potential fields [9], and potential fields have been used in robot soccer [10], [11]. Thurau et al. [12] 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 [13] we propose a methodology for creating a RTS game bot based on Multi-agent Potential Fields (MAPF). This bot was further improved in Hagelbäck and Johansson [14] and it is the improved version that we have used in this experiment.

IV. 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 Hagelbäck and Johansson [13]. 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.

A. Identifying objects

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

B. 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 Hagelbäck and Johansson [14]), 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 V-C.

C. 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]

a) The opponent units:

$$p(d) = \begin{cases} k_1 d, & \text{if } d \in [0, MSD - a[\\ c_1 - d, & \text{if } d \in [MSD - a, MSD] \\ c_2 - k_2 d, & \text{if } d \in]MSD, MDR] \end{cases} \quad (1)$$

TABLE I

THE PARAMETERS USED FOR THE GENERIC $p(d)$-FUNCTION OF EQUATION 1.

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

b) Own bases: Own bases generate a repelling field for obstacle avoidance. Below in Equation 2 is the function for calculating the potential $p_{ownB}(d)$ at distance d (in tiles) from the center of the base.

$$p_{ownB}(d) = \begin{cases} 5.25 \cdot d - 37.5 & \text{if } d <= 4 \\ 3.5 \cdot d - 25 & \text{if } d \in]4, 7.14] \\ 0 & \text{if } d > 7.14 \end{cases} \quad (2)$$

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

$$p_{ownU}(d) = \begin{cases} -20 & \text{if } d <= 0.875 \\ 3.2d - 10.8 & \text{if } d \in]0.875, l], \\ 0 & \text{if } d >= l \end{cases} \quad (3)$$

d) 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:

$$p_{sheep}(d) = \begin{cases} -10 & \text{if } d <= 1 \\ -1 & \text{if } d \in]1, 2] \\ 0 & \text{if } d > 2 \end{cases} \quad (4)$$

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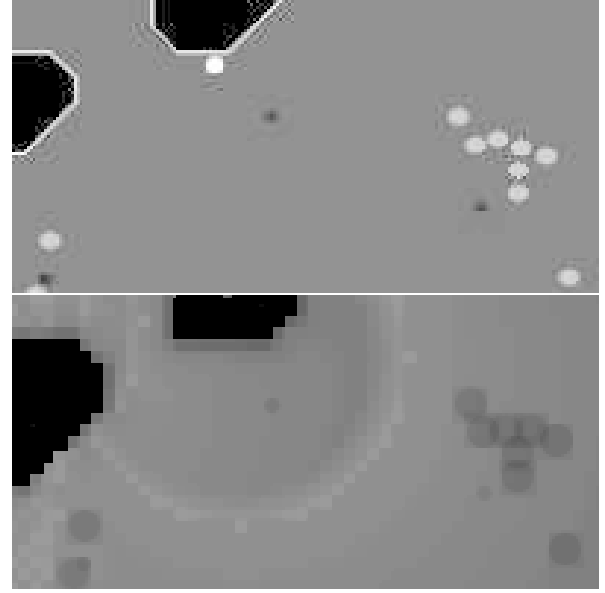

Fig. 1. Part of the map during a tankbattle game. The upper picture shows our agents (light-grey circles), an opponent unit (white circle) and three sheep (small dark-grey circles). The lower picture shows the total potential field for the same area. Light areas has high potential and dark areas low potential.

Figure 1 shows an example of a part of the map during a Tankbattle game. The screen shots are from the 2D GUI available in the ORTS server and a visualisation interface for potentials that we have developed. 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 own agents and the sheep.

D. 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.

E. 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.

F. 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.

1) 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:

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

(5)

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

(6)

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

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

(7)

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

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

(8)

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:

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

(9)

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.

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

(10)

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:

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

(11)

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.

2) 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

---

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

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:

$$\left(\sum_{k=0}^{m-1} a(k,i) \right) \cdot (damage_u - armour_i) >= HP_i \qquad (12)$$

$armour_i$ 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.

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.

V. 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 Hagelbäck and Johansson [14]. 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.

A. 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.

B. 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:

$$p_{terrain}(d) = \begin{cases} -10000 & \text{if } d <= 1 \\ -5/(d/8)^2 & \text{if } d \in]1, 50] \\ 0 & \text{if } d > 50 \end{cases} \qquad (13)$$

C. 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 14 below.
5) The total potential in (x,y) is the sum of the potentials each block generates in (x,y).

TABLE II

PERFORMANCE OF FOWBOT AND PIBOT IN 100 GAMES AGAINST FIVE
OPPONENTS.

Team	FoWbot			PIbot		
	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	—	—	—

$$p_{unknown}(md) = \begin{cases} (0.25 - \frac{md}{8000}) & \text{if } md <= 2000 \\ 0 & \text{if } md > 2000 \end{cases} \quad (14)$$

VI. 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.

A. 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 II 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.

B. 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 2 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 III, 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.
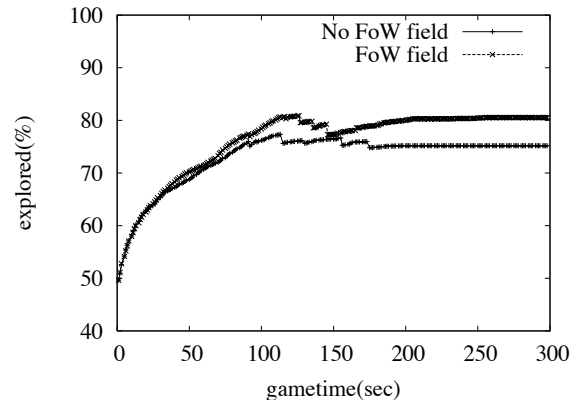

Fig. 2. The average explored area given the current game time for a bot using the field of exploration, compared to one that does not.

TABLE III

PERFORMANCE OF THE FOWBOT WITH AND WITHOUT FIELD OF
EXPLORATION (FOE) IN 20 MATCHES AGAINST NUS.

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

C. 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 3 shows the average time for both our bots in relation to number of own units left.

VII. DISCUSSION

The performance shows good results, but the question remains: could it be better without FoW? We ran identical
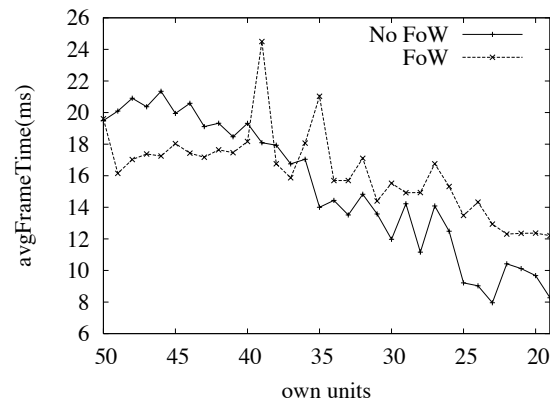

Fig. 3. The average frame time used for PIbot and FoWbot against team NUS.

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 II. 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 2 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 III) 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.

VIII. 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.

IX. ACKNOWLEDGEMENTS

REFERENCES

[1] Alexander Nareyek. Ai in computer games. *Queue*, 1(10):58–65, 2004.
[2] Michael Buro. ORTS — A Free Software RTS Game Engine, 2007. http://www.cs.ualberta.ca/~mburo/orts/ URL last visited on 2008-08-26.
[3] Michael Buro, Marc Lanctot, and Sterling Orsten. The second annual real-time strategy game ai competition. In *Proceedings of GAMEON NA*, Gainesville, Florida, 2007.
[4] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 5(1):90–98, 1986.
[5] R. C. Arkin. Motor schema based navigation for a mobile robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 264–271, 1987.
[6] J. Borenstein and Y. Koren. The vector field histogram: fast obstacle avoidance for mobile robots. *IEEE Journal of Robotics and Automation*, 7(3):278–288, 1991.
[7] M. Massari, G. Giardini, and F. Bernelli-Zazzera. 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*, 2004.
[8] J. Borenstein and Y. Koren. Real-time obstacle avoidance for fast mobile robots. *IEEE Transactions on Systems, Man, and Cybernetics*, 19:1179–1187, 1989.
[9] A. Howard, M. Matarić, and G.S. Sukhatme. 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)*, 2002.

[10] S.J. Johansson and A. Saffiotti. An electric field approach to autonomous robot control. In *RoboCup 2001*, number 2752 in Lecture notes in artificial intelligence. Springer Verlag, 2002.

[11] Thomas Röfer, Ronnie Brunn, Ingo Dahm, Matthias Hebbel, Jan Homann, Matthias Jüngel, Tim Laue, Martin Lötzsch, Walter Nistico, and Michael Spranger. GermanTeam 2004 - the german national Robocup team, 2004.

[12] C. Thurau, C. Bauckhage, and G. Sagerer. Learning human-like movement behavior for computer games. In *Proc. 8th Int. Conf. on the Simulation of Adaptive Behavior (SAB'04)*, 2004.

[13] Johan Hagelbäck and Stefan J. Johansson. 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)*, 2008.

[14] Johan Hagelbäck and Stefan J. Johansson. The rise of potential fields in real time strategy bots. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2008.