

## BACHELOR PROJECT ASSIGNMENT

**Student:** Stanislav Fifik  
**Study programme:** Open Informatics  
**Specialisation:** Computer and Information Science

**Title of Bachelor Project:** Automated Player for Game AI Challenge - Ants

### Guidelines:

AI Challenge is a series of competitions in programming agents for playing games. The exact task for the programs changes every year. In 2011, it is a game called Ants. Each player controls tens of units (Ants), which can be used for exploring the environment, gathering resources and fighting the opponents. The quality of the players is evaluated in tournaments.

1. The student will study the rules of the AI Challenge – Ants and the software libraries for creating the players for the game.
2. He will review the general algorithms and techniques used in games where players control larger amounts of units, mainly the potential fields and other techniques successful in the Open Real-time Strategy game.
3. Using the available libraries, he will implement a complete player that could be submitted to the competition.
4. He will compare the implemented player to the players successful in the 2011 AI Challenge competition.

### Bibliography/Sources:

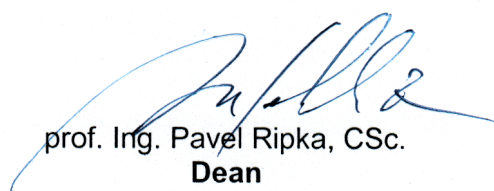
- [1] Balla, R. K., & Fern, A. (2009). UCT for tactical assault planning in real-time strategy games. Proceedings of the 21st international joint conference on Artificial intelligence (pp. 40–45). Morgan Kaufmann Publishers Inc.  
<http://www.aaai.org/ocs/index.php/IJCAI/IJCAI-09/paper/viewPDFInterstitial/632/587>
- [2] Hagelbäck, J., & Johansson, S. J. (2008). Using Multi-agent Potential Fields in Real-time Strategy Games. *Strategy*, (Aamas), 631-638.  
[http://ifaamas.org/Proceedings/aamas08/proceedings/pdf/paper/AAMAS08\\_0269.pdf](http://ifaamas.org/Proceedings/aamas08/proceedings/pdf/paper/AAMAS08_0269.pdf)
- [3] Zivan, R & Grinton, R & Sycara, K (2009). Distributed Constraint Optimization for Large Teams of Mobile Sensing Agents. (IAT), 347-354.  
[http://www.bgu.ac.il/~zivanr/files/MST\\_IAT\\_CR.pdf](http://www.bgu.ac.il/~zivanr/files/MST_IAT_CR.pdf)

**Bachelor Project Supervisor:** Mgr. Viliam Lisý, MSc.

**Valid until:** the end of the winter semester of academic year 2012/2013

  
prof. Ing. Vladimír Mařík, DrSc.  
Head of Department



  
prof. Ing. Pavel Ripka, CSc.  
Dean

Prague, January 9, 2012

## ZADÁNÍ BAKALÁŘSKÉ PRÁCE

**Student:** Stanislav F i f i k

**Studijní program:** Otevřená informatika (bakalářský)

**Obor:** Informatika a počítačové vědy

**Název tématu:** Hráč pro hru AI Challenge – Ants

### Pokyny pro vypracování:

AI Challenge je série soutěží zaměřených na tvorbu programů na hraní her. Každý ročník účastníci vytvářejí programy na hraní jiné hry. V roce 2011 to byla hra Ants, při níž hráč ovládá desítky jednotek (mravenců), které používá na průzkum prostředí, sbírání zdrojů a boj s ostatními hráči. Kvalita hráčů se vyhodnocuje na základě turnajů.

1. Student se seznámí s pravidly hry AI Challenge - Ants a softwarovými nástroji na tvorbu hráčů pro tuto hru.
2. Vytvoří si základní přehled o algoritmech a technikách používaných v hrách, kde hráč ovládá větší množství jednotek, především s potenciálový poli a dalšími technikami používanými v hráčích na hraní hry Open Real-time Strategy.
3. S použitím stávajících knihoven pro tvorbu hráčů naprogramuje jednoduchého, ale kompletního hráče, který může být přihlášen do soutěže.
4. Porovná implementovaného hráče s hráči úspěšnými v soutěži, která proběhla koncem roku 2011.

### Seznam odborné literatury:

- [1] Balla, R. K., & Fern, A. (2009). UCT for tactical assault planning in real-time strategy games. Proceedings of the 21st international joint conference on Artificial intelligence (pp. 40–45). Morgan Kaufmann Publishers Inc.  
<http://www.aaai.org/ocs/index.php/IJCAI/IJCAI-09/paper/viewPDFInterstitial/632/587>
- [2] Hagelbäck, J., & Johansson, S. J. (2008). Using Multi-agent Potential Fields in Real-time Strategy Games. *Strategy*, (Aamas), 631-638.  
[http://ifaamas.org/Proceedings/aamas08/proceedings/pdf/paper/AAMAS08\\_0269.pdf](http://ifaamas.org/Proceedings/aamas08/proceedings/pdf/paper/AAMAS08_0269.pdf)
- [3] Zivan, R & Grinton, R & Sycara, K (2009). Distributed Constraint Optimization for Large Teams of Mobile Sensing Agents. (IAT), 347-354.  
[http://www.bgu.ac.il/~zivanr/files/MST\\_IAT\\_CR.pdf](http://www.bgu.ac.il/~zivanr/files/MST_IAT_CR.pdf)

**Vedoucí bakalářské práce:** Mgr. Viliam Lisý, MSc.

**Platnost zadání:** do konce zimního semestru 2012/2013

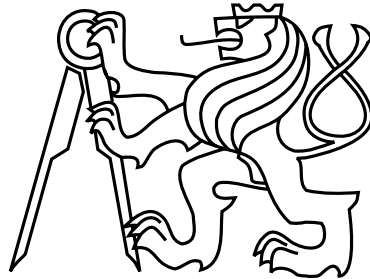
  
prof. Ing. Vladimír Mařík, DrSc.  
vedoucí katedry



  
prof. Ing. Pavel Ripka, CSc.  
děkan

V Praze dne 9. 1. 2012

Czech Technical University in Prague  
Faculty of Electrical Engineering  
Department of Computer Science and Engineering



Bachelor's Thesis

**Hráč pro hru AI Challenge - Ants**

*Stanislav Fífk*

Supervisor: Mgr. Viliam Lisý, MSc.

Study Programme: Open Informatics, Bachelor

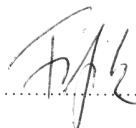
Field of Study: Computer Science

May 25, 2012

## Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 20. května 2012



.....



# Abstract

AI challenge is an international AI programming contest. The latest challenge (fall 2011), called Ants, was about searching food and enemies in two dimensional space, gathering food and destroying enemies by controlling large amount of units (ants). This thesis describes several algorithms that can be used in this domain, such as A\* search, potential fields, DCOP solvers, Minimax. It proposes solution for optimizing assignments based on distance, covering and exploring area using multiple agents and for dealing with combat situations.

Some of described algorithms were used by my player implemented for the tournament where appeared advantages and disadvantages of used potential fields thanks to the variance of the map set.

# Abstrakt

AI challenge je mezinárodní soutěž zaměřená na programování UI. Úkolem na poslední soutěži (listopad 2011), s názvem Ants, bylo hledání jídla a nepřátel v dvou rozměrném prostoru, zbirání potravy a likvidování nepřátel, to vše ovládním velkého množství jednotek (mravenců). Tato práce popisuje algoritmy použitelné na této doméně jako například A\* search, Potential fields, DCOP solver, Minimax. A předkládá řešení problémům jako jsou optimalizace přiřazování úkolů na základě vzdálenosti, pokrytí a prozkoumávání oblasti pomocí množství agentů a soubojů.

Některé z těchto algoritmů můj hráč přihlášený do této soutěže používal kde se díky různorodosti map projeví jak výhody tak nevýhody potential fields.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Game Specification</b>	<b>3</b>
2.1	Map	3
2.2	Turns and Phases	4
2.3	Endbot Conditions	4
2.4	Scoring	5
2.5	Cutoff Rules	5
2.6	Food Harvesting	6
2.7	Food spawning	6
2.8	Ants Hill Razing	7
2.9	Ant Spawning	7
2.10	Battle Resolution	7
<b>3</b>	<b>General Algorithm Description</b>	<b>9</b>
3.1	A*	9
3.1.1	A* code	10
3.2	Potential fields	11
3.3	DCOP	12
3.3.1	Problem definition	12
3.3.2	DCOP_MST solver	13
3.4	Minimax and its variations	16
3.4.1	Minimax	16
3.4.2	Alpha-beta pruning	17
<b>4</b>	<b>Problems and Solutions</b>	<b>19</b>
4.1	Food assignment and path planning using A*	19
4.2	Map exploring with potential fields	22
4.3	Map exploration as DCOP_MST	23
4.4	Simple combat	25
4.5	Better combat resolution using Minimax	26
4.6	Symmetry detection	26

<b>5</b>	<b>Testing</b>	<b>27</b>
5.1	Tournament . . . . .	27
5.2	Post-tournament updates . . . . .	27
5.3	Other bots . . . . .	27
5.3.1	xathis . . . . .	28
5.4	Results . . . . .	29
<b>6</b>	<b>Conclusion</b>	<b>31</b>
6.1	Evaluation . . . . .	31



# List of Figures

2.1	Sample game state with 2 players . . . . .	4
3.1	Ant not using potential fields (left). Ant using potential fields(right) . . . . .	12
3.2	Evaluated game tree with alpha-beta pruning . . . . .	17
4.1	Ants scattered around hill by MGM . . . . .	23



# List of Tables

5.1	Results of duels of my bot and other players from tournament . . . . .	29
-----	--	----



# Chapter 1

## Introduction

AI challenge [1] is great opportunity to test your skills in computer science and compare them with thousands of programmers worldwide in a tournament. This contest started by University in Waterloo Computer Science Club was open for public in 2010 when contest gain sponsorship from Google. For each challenge participants have to write program (bot) in almost any programming language that is able to plays a game versus an other's bots and upload the source code to contest server.

This fall (2011) started third public AI challenge called Ant's. The challenge was creation of bot controlling a lot of ants on an undiscovered grid with obstacles, food and enemies controlled by other bots. To survive in this dynamic and hostile environment, bot has navigate it's ants in order to explore map, find food and destroy enemies and their hills.

I wrote a bot focused on food gathering and map exploration to find more food, because for each gathered food a new ant is spawned. I tried to overpower enemies by count with just a simple combat strategy. My potential-driven bot using A\* for food assignment gained 948th place from 7897 total players. After tournament I further improved the bot and tested it against selection of bots from the tournament. The results shows that the food gathering was effective.

In Chapter 2 - Game Specification is described game environment, rules, goals. These specifications is a modified version available at [2].

In Chapter 3 - General Algorithm Description I'm proposing and describing algorithms with potential of solving problems on this domain.

In Chapter 4 - Problems and Solutions is described problems and ways to solved them using algorithms described in previous Chapter 3. This chapter also describes modifications to these general algorithms in order to improve their performance on this domain or to use them to solve slightly different problems than they're meant to.



## Chapter 2

# Game Specification

Ants is a turn-based multi-player strategy game set on a plot of dirt with water for obstacles and food that randomly drops. Each player has one or more hills where ants will spawn. The objective is for players to seek and destroy the most enemy ant hills while defending their own hills, scoring is described in appropriate section (Section 2.4). Players must also gather food to spawn more ants, however, if all of a player's hills are destroyed they can't spawn any more ants.

### 2.1 Map

The map is a grid of squares that wraps around at the edges (a torus). This means if an ant walks up across the top of the map they appear at the bottom, or walking to the right they appear at the left. A bot is a program that reads input about the squares it can currently see and outputs orders to move its ants around the map.

Figure 2.1 shows a sample map generated by included visualization script. The red and blue dots are ants, blue textured squares are water tiles, light brown squares are tiles with food and inside of the circles are hills.

Each ant (shown as red and blue dots on the map) can only see the area around (defined by view radius provided at the start of the game), so bots will not start with a full view of the map. This concept is in strategy games usually called Fog of War. Each turn the bot will be given the following information for all squares that are visible to its ants:

- a list of water squares, that have not been seen before
- a list of ants, including the owner
- a list of food
- a list of hills, including the owner
- a list of dead ants (from the last attack phase), including the owner



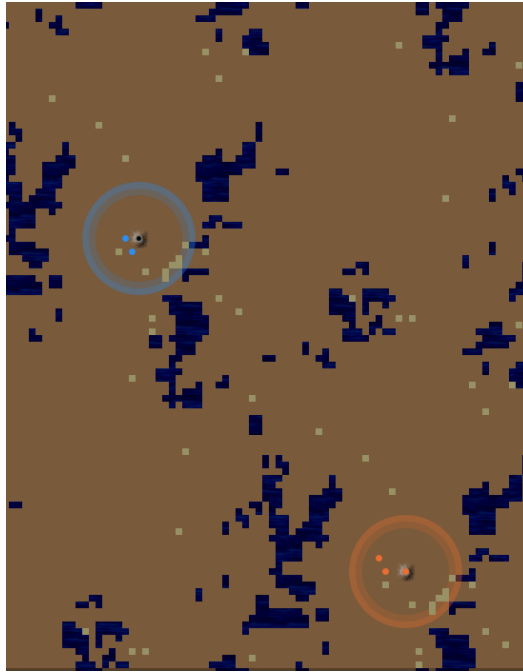


Figure 2.1: Sample game state with 2 players

## 2.2 Turns and Phases

A bot can issue up to one order for each ant during a turn. Each order specifies an ant by location and the direction to move it: North, South, East or West. Once the order is executed, ants move one square in the given direction.

The game then goes through 5 phases:

- move all ants (ants that collide in the same square are killed)
- attack enemy ants if within range
- raze ant hills with enemy ants positioned on them (Razing hill described later in Section 2.8)
- spawn more ants at hills that are not razed or blocked (Spawning described later in Section 2.9)
- gather food next to ants (food disappears if 2 enemies are both next to it)

After the phases, the bot will receive the next game state and issue more moves.

## 2.3 Endbot Conditions

Any of the following conditions will cause a player to finish participating in a game:

- The player has no live ants left remaining on the map.
- The bot crashed.
- The bot exceeded the time limit without completing its orders.

If a bot stops participating due to a crash or timeout, their ants remain on the board and can still collide and battle with other ants. Their ants just do not make any future moves and opponents are not explicitly told those ants' owners are no longer participating.

If a bot crashes or times out on a given turn then none of the moves received from that bot will be executed for that turn.

## 2.4 Scoring

The objective of the game is to get the highest score. Points are awarded by attacking and defending hills.

- Each bot starts with 1 point per hill
- Razing an enemy hill is 2 points
- Losing a hill is -1 point

## 2.5 Cutoff Rules

Games ends under following conditions.

**Food Not Being Gathered** If a game consists of bots that aren't capable of gathering food, then the game is cutoff. If the total amount of food is 90

**Ants Not Razing Hills** If a game consists of a dominant bot that isn't razing enemy hills, then the game is cutoff. If the total amount of live ants for the dominant bot is 90

**Lone Survivor** If there is only 1 bot left alive in the game, then the game is cutoff. All other bots have been completely eliminated (no ants on the map) or have crashed or timed out. Remaining enemy hills are awarded to the last bot and points subtracted from the hill owners.

**Rank Stabilized** If there is no bot with hills left that can gain enough points to gain in rank, then the game is cutoff. Even though bots without hills left could still possibly gain in rank, the game is not extended them, only those with hills. For each bot with a hill, it's maximum score (calculated assuming it could capture all remaining enemy hills) is compared to each opponents minimum score (calculated assuming it would lose all remaining hills). If any score difference can overtake or break a tie then the game continues. If no bot meets these criteria, the game is stopped.

**Turn Limit Reached** There is a maximum turn limit for each map. Each bot is given the limit. The game ends at this point.

## 2.6 Food Harvesting

Harvesting of food occurs each turn after the battle resolution process. If there are ants located within the spawn radius of a food location one of two things will occur:

- If there exist ants within the spawn radius belonging to more than one distinct bot then the food is destroyed and disappears from the game.
- If the ants within the spawn radius all belong to the same bot then the food is harvested and placed into the hive for that bot.

## 2.7 Food spawning

Food spawning is done symmetrically. Every map is symmetric, meaning each bot's starting position looks like every other bots starting position.

- Each game will start with a few food items placed within the bots starting vision, about 2-5.
- Starting food will be placed randomly on the map as well, symmetrically.
- Each game has a hidden food rate that will increase the amount of food in the game. Then the amount to be spawned is divisible by the number of players, then that amount of food will spawn symmetrically.
- The entire map is divided into sets of squares that are symmetric. The sets are shuffled into a random order. When food is spawned, the next set is chosen. When all the sets have been chosen, they are shuffled again.
- Every set will spawn at least once before a set spawns a second time. This means if you see food spawn, it may be awhile before it spawns again, unless it was the last set of the random order and was then shuffled to be the first set of the next random order.
- Sometimes squares are equidistant to 2 bots. This makes for a set that is smaller than normal. The food rate takes this into account when spawning food.
- Some maps have mirrored symmetry so that a set of symmetric squares are touching. It would be unfair to spawn so much food in one place, so these sets are not used. If you can find a mirror symmetry after exploring the map, then you can avoid those spots when looking for food.

## 2.8 Ants Hill Razing

The objective of the game is to raze your opponents hills and defend your own hill. A hill is razed (destroyed) when an enemy ant is at the same location as the hill after the attack phase. Razed hills do not spawn ants anymore. If all your hills have been razed, but you still have ants, your bot is still alive and your ants can still move, attack, gather food and raze hills.

## 2.9 Ant Spawning

As food is harvested, it is placed in the hive. Each food will spawn 1 ant. Ants are only spawned at hills.

- The hill must not have been razed.
- The hill must not be occupied by an ant.

Only 1 ant can be spawned on a hill each turn.

For maps with multiple hills, 1 ant can be spawned at each hill if there is enough food in the hive. If there is less food in the hive than there are hills, each hill is given a priority. The last hill to have an ant on top is chosen last or the hill to have been touched the longest ago is chosen first. In case of a tie, a hill is chosen at random.

This means that if you always move ants off of the hill right away, the spawned ants should be evenly spread between the hills.

You can control which hill to spawn at by keeping an ant nearby to block the hill when you don't want it to spawn ants.

## 2.10 Battle Resolution

If ant appears in attack radius of other ant of different color the battle begins. The battle resolution is based on ant distraction, which sum of enemy ants within ant's attack range. Because the ants less distracted are considered to be more focused to kill enemy without getting killed. Equally distracted enemy ants kills each other. The ant kills each ant within its attack range that is more or equally distracted than it is. Resolution starts with most distracted ants so ant can kill and be killed in same turn. It also can kill more ants in single turn.

Ants doesn't care about enemies colors. Distraction is calculated as number of ants that are not the same color as the ant.

For example let's consider case where we have two ants of different color within each other's attack range. Both ants will have distraction value one and so both will die.

More interesting case is when there is two ants of same color (red) and one of other (blue), also each of them within each other's attack range. Both red ants has distraction value one while blue one has distraction value two and thus the blue one will die while both red ants survive.



## Chapter 3

# General Algorithm Description

This chapter describes algorithms that can be used on this domain. While implementing the player for tournament, I focused mainly on food gathering to win game by overpowering enemy by count. The following algorithms were chosen to solve this problem: A\* (Section 3.1), Potential Fields (Section 3.2), DCOP solvers (Section 3.3).

This chapter also contains fragments of code used as illustration described algorithms. Please note that instead of pseudo-code is used python.

### 3.1 A\*

A\* search algorithm as described in [?] is informed, best-first, path finding algorithm which finds a least-cost path on graph. It uses optimistic heuristic function (usually denoted  $f(x)$ ) to first search the routes that appear to be most likely to lead towards the goal. It's sum of two other functions. The path-cost function (usually denoted  $g(x)$ ) and heuristic estimate (usually denoted  $h(x)$ ). The path-cost function is cost from starting node to current node

**Heuristic function** is estimate of distance from current node to goal. This function must be admissible heuristic. It means that estimated distance must not be more than real distance, other way the found path is not guaranteed to be shortest. If the heuristic function is monotone (difference of heuristic values of any two neighboring nodes is not more than cost of edge between these nodes) algorithm doesn't need to reprocess any of already processed nodes and a closed set can be used. It's possible because the  $g(x)$  for every opened node is always the lowest possible while using monotone heuristic function.

Summing these two functions we get estimated cost of path through processed node ( $f(x)$ ). Using  $f(x)$  we can sort our open list and use it as priority queue.

**Performance** heavily depends on quality of heuristic function ( $h(x)$ ). If it's very close to the true cost of the remaining path fewer nodes has to be opened and efficiency will be high. As the quality of heuristic function goes down, the A\* acts more like bread-first search as the bread-first search is A\* with underestimated heuristic function ( $h(x) = 0$  for all nodes).

### 3.1.1 A\* code

Following code (Listing 3.1) shows generic A\*.

```

1 def a_star(start, goal):
2     closedset = []
3     came_from = {}
4     g_score = {start:0}
5     h_score = {start:h(start, goal)}
6     openlist = PriorityQueue([start], key=lambda x : g_score[x] + h_score[x])
7     while openlist:
8         current = openlist.poll()
9         if current is goal:
10            return reconstruct_path(came_from, goal)
11        closedset.append(current)
12        for neighbor in current.neighbors():
13            if neighbor in closedset:
14                continue
15            tentative_g_score = g_score[current] + edge_cost(current, neighbor)
16            if neighbor not in openlist or tentative_g_score < g_score[neighbor]:
17                came_from[neighbor] = current
18                g_score[neighbor] = tentative_g_score
19                f_score[neighbor] = g_score[neighbor] + h_score[neighbor]
20                if neighbor not in openlist:
21                    h_score[neighbor] = h(neighbor, goal)
22                openlist.push(neighbor)
23    return None
24
25 def reconstruct_path(came_from, current_node):
26     if current_node in came_from:
27         return reconstruct_path(came_from, came_from[current_node]) + [current_node]
28     else:
29         return current_node

```

Listing 3.1: Generic A\* code

**Inicialization** Algorithm starts by defining several sets and maps. Maps holding  $g$  and  $h$  values (line 4 and 5) are used to keep track of distance from start to each node and heuristic estimate from each node to goal respectively. Open list is used to figure out what node should be processed next. Order is defined by  $f$  value of node which is defined as sum of  $g$  and  $h$  values (key on line 6). Using prioritized queue with ordering key set to  $f(x)$  is usually good way to do it (line 6). To keep track of already processed nodes the closed set is used (line 2). To reconstruct found path we need to know parent node for each probed nodes ( $came\_from$  map on line 3). Parent node is adjective to current node while having lower distance to start.

**Main loop** The algorithm loops while the open list is not empty. If the open list is empty at this point it means that the path could not be found. During each iteration polls out the head of open list, which is a node with lowest  $f$  value and checks it for goal. If the goal is reached, path is reconstruct and returned otherwise the processed node is closed by adding it to closed set and it's adjective nodes that are not in closed set are added to open list. To add node to open list the  $f$  value has to be known. The  $g(x)$  is calculated by adding cost of the edge between processed node and it's adjective node to  $g$  value of processed one. The  $h$  value is result of heuristic function. If the node is already in open list we should check



whether it's new g value is lower than the one calculated earlier, if so we need to update it's f value and position in open list.

## 3.2 Potential fields

Potential fields method generates artificial "electromagnetic" field from points of interest (goals, obstacles, enemies) and navigates unit through this field like charged particle. This method doesn't need discretized space in contrast to most path finding algorithms. It also acts reactively and thus it can easily respond to changes of environment. However the reactive nature can lead to local optima problem.

In artificial potential field methods, an obstacle is considered as a point of highest potential, and a goal as a point of lowest potential. Navigated unit always moves from a high potential point to a low potential point. Algorithm sets potential field function and then repeats selecting reachable point with minimum potential and navigating unit to this point until the goal is reached.

The potential fields driven ORTS bot had great success when playing against winner of 2007 ORTS tournament ([3] and [4]). It can be used for generating soft edge of obstacles, navigating cluster of units or keeping unit slightly out of enemy unit's attack range.

Potential fields is fast and simple path finding substitution in dynamic game worlds because units doesn't need to do full path search and can look just one step ahead instead. Generating full path on large distances can be huge waste of resources in dynamic environment because the path can become obsolete because of changes in the environment. It has very nice results on open map where obstacles are or can be fill to convex shapes. Unfortunately this reactive method leads to local optima problems and makes it unusable in complicated environments like mazes. But it can be used as supporting method for more sophisticated path planning. For example the path of group of units can be found by A\* and following the path can be done by attracting units to segments of this path. Adding the obstacle potential will force units to avoid them.

Following two images demonstrate advantages of potential fields on open maps. This map was used in AI challenge tournament (it's name is `random_walk_p02_02`). In both cases agents start at yellow node and try to reach green node, black nodes are obstacles and darkness of node in Figure 3.2) displays its potential. The potential values has been set as sum of one quarter of potential of adjacent nodes. Starting with 0 potential of passable nodes and 1 of obstacles.

Both agents choose next move to minimize Manhattan distance to goal, but agent using potential fields choose node with lower potential. As you can see the potential-aware agent (Figure 3.2 right) can easily pass obstacles with small gaps then the one without potential-awareness (Figure 3.2 left).

Unfortunately there is a lot of cases in AI challenge Ants problem where both reactive agents fails and path searching algorithm has to be used to successfully navigate agents.

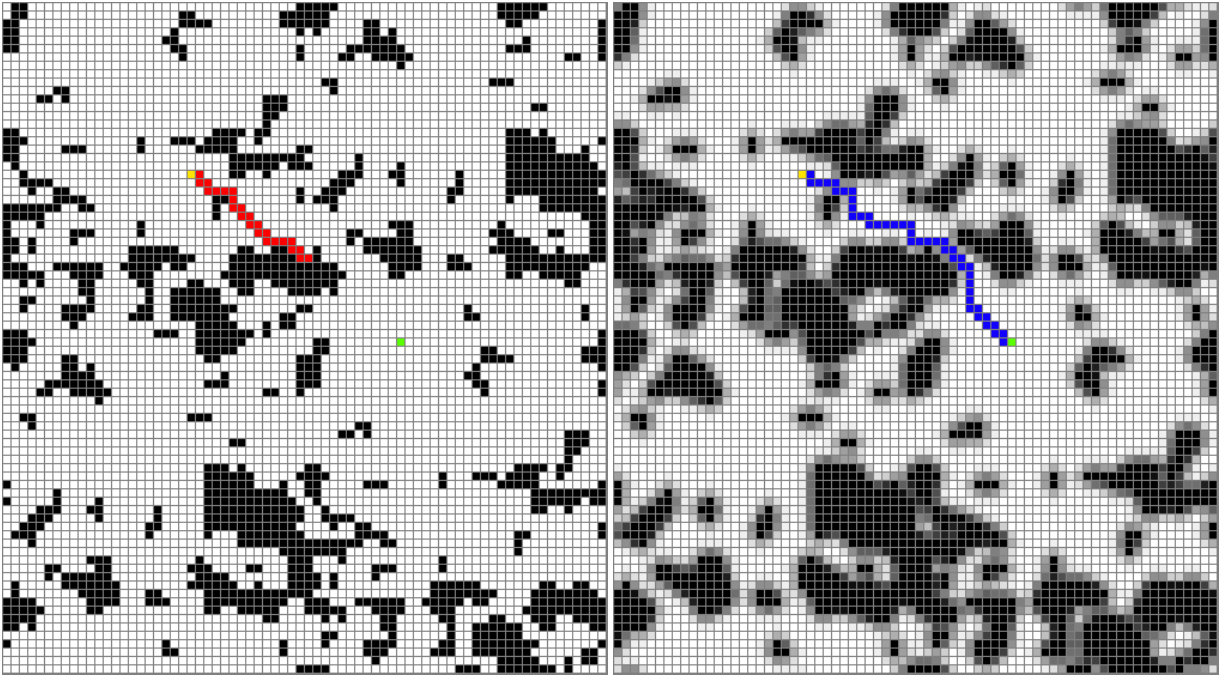


Figure 3.1: Ant not using potential fields (left). Ant using potential fields(right)

### 3.3 DCOP

#### 3.3.1 Problem definition

DCOP (Distributed constraint optimization Problem described in [8]) is problem where group of agents must distributively choose values for a set of variables to either minimize or maximized the cost of constraints over the variables.

DCOP is defined as a tuple  $(A, X, D, R)$ , where:

- $A$  is set of agents
- $X$  is set of variables
- $D$  is set of domains
- $R$  is function that maps every possible variable assignment to a non-negative cost.

The objective in a DCOP is to have each agent assign values to its associated variables in order to either minimize or maximize sum of cost's for a given assignment. DCOP is general model that can be used to represent many real world problems as Meeting Scheduling and Supply Chain Management.

To use DCOP for managing surveillance the adaption for Mobile Sensing agent Teams (DCOP\_MST) described in [6] can be used. In DCOP\_MST the agents adjusts their positions to adapt to the dynamically changing environment and the dynamic changes in the quality of information reported by sensors. Environment changes can be caused by change of

priorities of area, unit movement. Quality of information can be changed by sensor failure and whether changes. A MST problem several challenges not met by standard DCOP model. Agents will need position and because of technology limitation the sensing and mobility range has to be set. By limiting the mobility range a set of alternate position is defined. Alternate position and sensing range defines affected agents by agent's move. One of most challenging adaptation in the design of DCOP\_MST is generation of goal function which has to characterize the dynamic nature of the environment and dynamic quality of agent's reports. The quality of agent's reports (credibility) is calculated using reputation model. The required coverage for each point is represented total sum of agent's credibility to necessary successfully cover it. The model's goal function is difference between credibility of agents covering the area and coverage requirement. Because of the dynamic nature of the problem a complete algorithm would not be practical. Some of existing local search algorithms for DCOP can be adjusted to solve a DCOP\_MST. Algorithm based on the Maximum Gain Message (MGM) can be used in order to solve DCOP\_MST. It is distributed message passing local search algorithm with quick convergence. It's also monotonic and thus should minimize expensive agent movement. Being incomplete and monotonic causes convergence to a local minima.

DCOP\_MST bring few new concepts not used in standard DCOP. The first one is agent's position, which has to be represented by coordinates. Then two ranges has to be defined, the Sensing Range(SR) of agent, which represent coverage of the agent, and Mobility Range (MR) which represent range accessible in a single iteration. Using newly defined Mobility Range the Domain set can be redefined. Domain of Agent includes all alternative positions within agents Mobility Range from agents current position. Based on reputation model the credibility variable for each agent can be compute as real positive number. To represent coverage requirement for each point we need to define environmental requirement function (ER), which express required sum of credibility variables of agents with agents within sensing range. Function current difference (Cur\_Diff) defined for each point returns the difference between value of environmental function (ER) and sum of the credibility variables of agents currently covering it.

The global goal of agents is to cover area so the largest value of current difference is zero. This can not be always achieved, so more general goal is needed, which is minimizing value of current difference function over all points.

A ordered set OE including all types of events ordered accordingly to their chronological order. Each event in OE includes it's type, time of occurrence and its location (in case of an environmental event) or the agents involved (in case of an event which affects agents' credibility). Each agent can send message to each of the other agents.

In DCOP neighboring agents are the agents that can be influenced by an assignment change, because in the DCOP\_MST the agents are aware of their position and their movement is limited two agents are considered neighbors, if after move towards each other, their sensing ranges overlap. Neighboring is also dynamic aspect of DCOP\_MST.

### 3.3.2 DCOP\_MST solver

In [6] is proposed a solving algorithm for DCOP\_MST. The proposed algorithm choose local search for solving problems instead of complete search because of dynamic nature of

environment and limited mobility of agents. The MGM (Maximum Gain Message) is local search algorithm selected for this solver for it's fast convergence and simplicity. It's monotone and thus it would avoid redundant movement of agents. In MGM after agent's receives the assignment of all its neighbors, they compute maximal improvement (cost reduction) to its local state possible by changing its assignment and sends this proposed reduction to its neighbors. After collecting the proposed reductions from its neighbors agents changes its assignment only if its proposed reduction is grater than the reduction proposed by all of its neighbors.

```

1 def mgm_mst(agent):
2     while True:
3         agent.send_to_neighbors(agent.cur_position)
4         positions = agent.recieve_neighbors_positions()
5         local_reduction = best_possible_local_reduction(agent)
6         agent.send_to_neighbors(local_reduction)
7         local_reductions = agent.recieve_neighbors_lr()
8         if (local_reduction["value"] > 0):
9             if (local_reduction["value"] > max(local_reductions, key=lambda x:x["value"]):
10                agent.cur_position = local_reduciton["position"]
11
12 def best_possible_local_reduction(agent):
13     possible_pos = agent.possible_positions()
14     temp_diff = cur_diff.subtract_coverage(agent.coverage)
15     new_pos = select_position(possible_pos, temp_diff, agent)
16     cur_covered_points = temp_diff.points_within_area(agent.cur_position, agent.SR)
17     new_covered_points = temp_diff.points_within_area(new_position, agent.SR)
18     cur_coverage = max(cur_covered_points - new_covered_points)
19     new_coverage = max(new_covered_points - cur_covered_points)
20     return {"position" : new_pos,
21            "value" : min(cur_coverage - new_coverage, agent.credibility)}
22
23 def select_position(positions, func, agent):
24     if len(positions) == 1:
25         return positions.pop()
26     visible_set = union([func.points_within_area(pos, agent.SR) for pos in positions])
27     target_set = [point for point in visible_set
28                  if point == max(visible_set) and point > 0]
29     if not target_set:
30         return random.choice(positions)
31     if not [pos for pos in positions
32            if all([points.distance(pos) <= agent.SR for point in target_set])]:
33         target_set = max([points for points in target_set if point.distance(pos) <= SR
34                          for pos in positions], lambda x : len(x))
35     possible_pos = [pos for pos in positions if all([points.distance(pos) <= agent.SR
36            for point in target_set])]
37     intersection_area = intersection(
38         [func.points_within_area(pos, agent.SR) for pos in positions])
39     new_func = func.subtract_coverage(intersection_area)
40     return select_pos(possible_pos, new_func, agent)

```

Listing 3.2: DCOP\_MST solver

Each agent runs it's loop of `mgm_mst`. In which the agents sends and receives their current positions to and from their neighbors (lines 3 and 4). Then calculates it's best possible local reduction (line 5), and exchanges it with it's neighbors. Then changes their position to position selected by local reduction if their positive local reduction value is larger then reductions proposed by their neighbors (lines 8–10).

To select best possible local reduction appropriate function is used. This function gets agents possible positions (those in its MR) and creates `Cur_Diff` without agents coverage

(Cur\_Diff where agent is not active). Then new position is found and it's value calculated as difference of two points with highest value, one from set of points that won't be covered from new position but are covered from current and one from set of points that is not covered now but will be covered from new position. As the agent can't provide higher coverage that is it's credibility value is reduced to it's credibility if higher.

To select new position, recursive function is used. This function returns position if only one position is available otherwise it finds all points from passed diff (func) that is within agent SR from any available position and all selects positive points with highest value. If none of the points is positive any position can be return. Now we want to find positions that covers all these points (line 35), this may not be always possible (check on line 31) so we try at least find positions that covers largest subset (line 33). As we found positions that is best to use we can consider points within SR from all these positions covered and update diff(func) and finally recursively call this function.

## 3.4 Minimax and it's variations

### 3.4.1 Minimax

Minimax (described at [7]) is algorithm for zero-sum and usually two player games, which minimizes the maximum possible loss. Algorithm recursively explores game tree and returns next move that minimizes loss. Because of depth and branching factor of game trees in most games, it's not possible to compute complete tree. This is solved by limiting exploring by depth and using heuristic function to estimate how good it would be for player to reach that state (node of the game tree).

The following Listing 3.3 describes generic minimax function. This function uses gain instead of loss, which is usually used, and thus it's maximin (maximizing minimum gain) but the idea is same. The algorithm is started for each possible moves with game state after that move, depth we want to compute to and set to maximize.

```
1 def minimax(state, depth, minimize = False):
2     if depth == 0 or state.is_final():
3         return state.heuristic_gain()
4
5     alpha = Inf if minimize else -Inf
6
7     for child in state.childs():
8         score = minimax(child, depth-1, not minimize)
9         alpha = min(alpha, score) if minimize else max(alpha, score)
10    return alpha
```

Listing 3.3: Minimax code

Each call of minimax function returns either heuristic evaluation of processed state in case that desired depth was reached or the state is final (lines 2 and 3 of Listing 3.3) or minimum/maximum value of minimax called on it's children (lines 7 to 9). The minimizing flag is flipped on each depth (call of minimax on line 8). This way the algorithm simulates opponent's best possible move, the one that minimizes our gain (and because it's zero-sum game, maximize his gain).

### 3.4.2 Alpha-beta pruning

The original minimax can be speed up by not expanding and calculating subtree which can not change the result. Overview of this method is at [7]. To find out which nodes doesn't need to be expanded alphabeta pruning denotes two new variables alpha and beta, which is used as upper and lower bound.

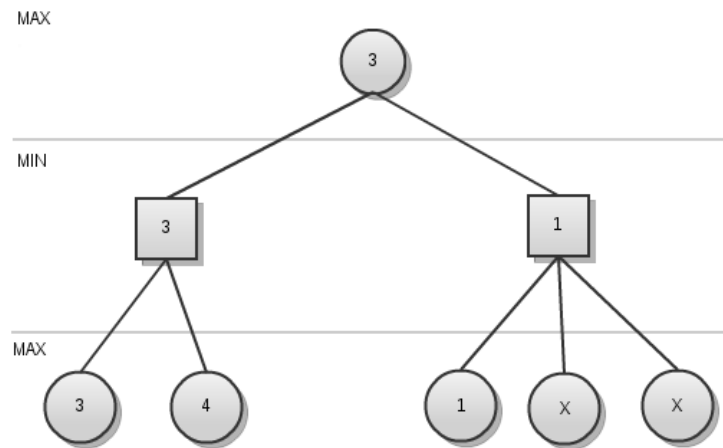


Figure 3.2: Evaluated game tree with alpha-beta pruning

The Figure 3.2 shows possible pruning. Leaves marked with X (possibly some huge subtrees) doesn't need to be evaluated because the right subtree (with value 1) won't be selected as maximum by root node. We know that as soon as we find the leaf with value 1, because value of parent can be now only same or lower than 1.

```

1 def alphabeta(state, depth, minimize = False, alpha = -Inf, beta = Inf):
2     if depth == 0 or state.is_final():
3         return state.heuristic_gain()
4
5     for child in state.childs():
6         score = minimax(child, depth-1, not minimize, alpha, beta)
7
8         if minimize:
9             beta = min(beta, score)
10        else:
11            alpha = max(alpha, score)
12
13        if beta <= alpha:
14            break
15
16    return beta if minimize else alpha

```

Listing 3.4: Alphabeta code





## Chapter 4

# Problems and Solutions

Following chapter describes usage of algorithms described in Chapter 3. It shows how to use these algorithms in order to navigate ants, assign targets and explore map. It also describes few performance improvements that we possible thanks to game specifications.

### 4.1 Food assignment and path planning using A\*

Following section shows domain specific modifications to generic A\* algorithm that I figured out in order to make it more effective. Ants are played on a grid where some nodes are inaccessible and food which randomly appears on accessible nodes has to be gathered by ants. There is a lot of ants and a lot food, so ant has to be assigned to food and a path has to be found. Optimal assignation would be when sum of distance necessary to overcome by ants in order to gather all food is minimal and thus the number of turn is minimal as well. I used A\* for both, the food assignment to the closest ant and path finding.

Because of movement limitations in this domain the Manhattan distance is good choice of heuristic function. It is admissible and even monotone. A\* is usually used to find the shortest path from one start to one goal. If we want to use A\* to navigate ants in order to collect food we need to figure out which ant should collect which food. Fortunately the A\* can be modified to do this ant to food mapping for us and even provide us the shortest path in a single run. In order to collect food as soon as possible we're looking for ant which is closest to food. To find it we need to run A\* from food and our ants as goals. We also need to modify the heuristic function. Instead of Manhattan distance we have to use minimum of all Manhattan distances from node to each ant (Listing 4.1).

```
1 def h(start, goals):  
2     return min([manhatan_distance(start, goal) for goal in goals])
```

Listing 4.1: Modification of heuristic function for multiple goals

As we are using lowest value for set of admissible heuristics the new heuristic stays admissible. To prove it is monotone we need to prove that it keeps monotone when goal with minimum distance is changed between adjective nodes because as far as we keep using same goal the heuristic acts as standard Manhattan. Heuristic is monotone when difference of values of heuristic function of and two adjective nodes is not larger than cost of edge

between these nodes. Cost of all these edges is in our case one. Value of heuristic function is based on selected goal. Let's say we have two adjective nodes A and B and two goals C and D and heuristics  $H(x) = \min(h(x,g))$  where x is node, g is set of all goals and  $h(x,g)$  is Manhattan distance from x to g.  $H(A) = h(A,C)$  and  $H(B) = h(B,D)$ . This means that  $H(A) \leq h(A,D)$  and  $H(B) \leq h(B,C)$ . And because h is Manhattan distance and because of the movement limitations:  $\forall g \in Goals : |h(A,g) - h(B,g)| = 1$ .

$$H(A) \leq h(B,D) + 1$$

$$H(A) - H(B) \leq 1$$

Instead of path reconstruction from food to found closest ant we can save the path-cost ( $g(x)$ ) to nodes and it will allow ant follow path without full path reconstruction. This also make it reusable in future and allow ant to slightly change it when obstacle appears without need to run new search. Ant follow path by choosing adjective, passable node with smallest distance to goal.

If we change  $h(x)$  to select value saved in nodes instead of Manhattan distance when possible we can improve future searches performance. We can even stop search when node with known path to goal appears on head of open list and navigate ant through this node. It's possible because the open list is sorted by  $f(x)$  which value can not be lower that real path length as so we know that all other possible path through other nodes in open list won't be better then this one.

If the closest ant isn't satisfying solution (it has already assigned more important task) the algorithm can be resumed. All it takes is removing the ant from goals and recalculating  $h(x)$  values for all nodes in open list and reorder it by updated  $f(x)$  values. We can do this because the heuristic function is monotone and so we can be sure that all already calculated  $g(x)$  values are correct. We just need to fix wrong order of open list which may be broken because of possible changes of  $H(x)$  values (Only nodes which used removed goal for calculating minimum estimated distance has wrong  $f(x)$  values and thus are in wrong order). This search resuming can be very useful when the second closest ant is close to first one. Smart way to achieve this is turning A\* to iterator. In the Listing 4.2 is shown python way to do so by modifying 10th line of original A\* (Listing 3.1).

```

1 yield goal
2 goals.remove(goal)
3 recalculate_openlist(openlist, goals, goal)
4 continue

```

Listing 4.2: Switching to iterator my modifying 10th line of Listing 3.1

To assign as much food as possible to our ants A\* for each food has to be run. Order in which the searches are run may lead to different results, based on how we solve assigning food to ant that already has assigned food.

If we decide to change its assignment when the new one is strictly better than the previous one then we should rerun search for the previous one. This can be easily accomplished by creating queue of all food and pushing there food that lost its assignee. We can take advantage of resuming the searches, described above, by keeping them in memory till we solve complete queue.

Other way to do this is creating priority queue of the searches and processing one A\* cycle of the one with lowest  $f(x)$  value of node at the head of it's open list repeatably until all of them has assignment. This way the searches can share goals so the found solution in one of them will remove this solution from goals of all searches and force them to recalculate their open list.

Open list recalculation can be accelerated by memorizing what goal was used in  $h(x)$  function (Listing 4.3). Only nodes where the goal, that has been just removed, was used in heuristic function has to be recalculated. This can have great impact on performance using the method with simultaneously processed searches.

```

1 def h(start , goals ):
2     ( distance , node)=min ([[manhatan_distance(start , goal), goal] for goal in goals])
3     h_chosen_goal[start]=goal
4     return distance
5
6 def recalculate_openlist(openlist , goals , removed_goal):
7     affected=[node for node in openlist if h_chosen_goal[node] is removed_goal]
8     for node in affected:
9         openlist.remove(node)
10        h_score[node] = h(node, goals)
11        openlist.push(node)

```

Listing 4.3: Accelerated open list recalculation

We should also take in mind that game is dynamic and food that is too far from our ants may be gathered by enemy or closer ant without without assignment can appear (spawned or one that just finished it's task). Using A\* for planning path on long distances can use a lot of resources, the resource consumption grows exponentially. Setting maximum distance cutoff isn't bad idea. Reasonable value is somewhere between one and two view distances. Cutoff is applied when currently processed node's  $f(x)$  value is more then the set maximum distance or to save memory we can discard these nodes instead of putting them to open list.

Setting cutoff has another positive effect. We can remove goals theirs  $h(x)$  value at the beginning of the search is greater then our maximum distance. This speeds up further  $h(x)$  evaluation.

## 4.2 Map exploring with potential fields

While Fog of War concept and food is spawn randomly we need to keep informations about each map tile as new as possible in order to discover need food. To do so, the potential fields can be used. To navigate ants to tiles with old information we need to set new variable for each map tile, which will represent how long the tile has not been seen. Using this negation of this value as potential we can navigate ants toward lowest potential and make them discover map. While the visible or water tiles has 0 potential and other tiles has negative potential and longer it was unseen the lower it is, we can navigate them by selecting direction where sum of unseen value of tiles slightly out of visible range in that direction is lowest.

I found this technique to be quite successful on open maps but had problems on more complex maps like mazes. The ants stuck close to wall trying to reach field that is behind it.

To prevent this we can run breadth-first searches (BFS) through non-water tiles limited to distance of visible range for every possible position and calculate the potential as sum of potential of all these tiles. This method is quite resource heavy. It involves five breadth-first searches and sum of a lot of tiles' potential.

Problem with running a lot of searches can be solved by saving the result to tiles and it can even improve A\* performance if the distances generated during BFS are saved as well. The problem of huge sum where most of values is zero can be also solved. We need to save tiles that on the edge of the search. Tiles which straight line distance to center tile is less than view distance and more then view distance - 1.

It can be tweaked even more by splitting the edge into 4 parts and return only 2 appropriate parts based on direction where from we came but we already removed most of unnecessary tiles and kept the potential evaluation unchanged. This tweak would remove just another half of nodes and make the evaluation function more complex.

During testing bot on maze with right angle corridors appeared problem with trying to turn too soon. Since the area behind the is not discovered, the ant is attracted there as soon as the BFS can pass around the end of wall. This situation can be solved by reconstructing the path generated by previous BFS to point that is just behind the wall and following it. We can also keep setting direction changers on tiles we follow as long as the Manhattan to the point behind the wall increases. This direction changers will change selected direction of any further bot stepping on this tile and trying to repeat same mistake.

Even though the ants stopped trying to explore unreachable tiles the closed map problems wasn't solved. On these closed maps there is usually more ants on smaller area and if everything around the ant is visible, the ant doesn't know where to move.

To solve this I wrote simple workaround just before the tournament. I assigned randomized simple fallback instructions to each ant. The instruction contained list of preferred directions and ant which didn't knew where to move selected first possible instruction from this list. The list was left or right rotating with random first direction.

Better solution would be calculating potential vector from it's neighbors (neighbors are ants with overlapping view ranges), which is implemented in the newer version. To simplify calculation I just selected the closest neighbor.

### 4.3 Map exploration as DCOP\_MST

As DCOP solvers usually leads to static solution and thus make agent to find local optima and stay there. To force agent to explore we need to dynamically change the ER (environmental requirement) function to return values that depends on duration for which the area wasn't covered. In Ants we want to cover every accessible area of map (non-water tiles) and so we can define ER function like:  $\min(\text{currentTurn} - \text{lastSeenTurn}, \text{threshold}) / \text{threshold}$  where  $\text{currentTurn}$  and  $\text{lastSeenTurn}$  is values of turn counter now and when the area was last seen and  $\text{threshold}$  is number of turn after which the ER function is maximal. Note that this function returns values from 0 to 1. To fully discover the any tile on map it needs to be covered at least by one ant and there is no difference between ants so reputation model proposed in original algorithm is not necessary and we can simply consider that credibility of all ants is one. This and the fact, that to successfully cover any area in Ants is only one ant needed, are reasons why ER function values is in range 0 and 1.

Changing ER function for some important positions like hill or front lines can create guards or send more ants to fight and help holding the area. The ER represents how many ants we want assign to cover some area.

Unfortunately I wasn't able to implement the solving algorithm that would work fast enough. The algorithm loops until there is no cost reduction, while in each iteration runs local reduction for each ant and during each local reduction is a lot of iterations over tiles within the sensing range (approximately 200 tiles) of each possible position. My implementation was able to operate with 20 ants while the turn time limit was increased to 3 seconds which was 3 times more than the tournament time limit. This slowness made it impossible to compete with other players.

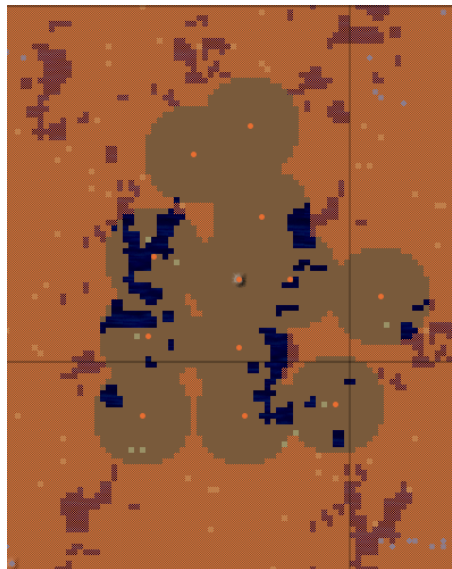


Figure 4.1: Ants scattered around hill by MGM

Although it was impossible to test it with larger amount of ants, the behaviour was good. Instead of rapid exploring the bot covered large area around hill with just a few ants and

when new ant was spawned ants at borders moved to cover new area. The coverage is shown on Figure 4.3, the part map with red overlay is out of sensing range. This behaviour can be used to build defensive borders by setting larger environmental requirement to positions inside visible ranges of border ants.

To speedup the algorithm I changed next move selecting function to selecting tile which has highest sum of current difference of tiles in its sensing range, but it affected the performance just slightly, because the main problem is in large amount of operations on current difference maps (500,000 with 15 ants every turn) which I wasn't able to significantly decrease.

I'm convinced that this method is good way to solve exploration problem because unlike potential fields we can set number of ants (not just attraction) we want to keep close any point. Thanks to the communication between ants there would be less unnecessary moves, like two ants moving to explore almost same area.

There is also one significant drawback of this method when comparing to potential fields. Unlike when using potential fields where we can get all possible moves ordered by their value this way we can get only one next move for each ant, which means that other methods like safe tiles checking (described in following section) has to be implemented as input for the algorithm and modify set of possible moves, before the algorithm starts. For example the safe tiles checking can be in complicated situations slow and it have to be always run for each possible position.

## 4.4 Simple combat

My tournament player was focused mostly on food gathering and tries to win with overpowering opponents by count. But algorithm for solving combat situation was necessary. This simple algorithm evaluates fields as dangerous or save based on opponent position. Then ants are allowed to move only to save fields. There is no inter-agent cooperation and there is a lot of special cases where this algorithm fails. My tournament bot used more simple method and didn't calculated with how many enemies the enemy ant is fighting with. This method was implemented after the tournament.

To figure out which fields are dangerous we need to know for each field on map how many enemies may have the field in it's attack radius next turn and how many ant's are attacking. To do so we can use following algorithm. (Listing 4.4).

```

1 influence = {owner:{field:0 for field in Map} for owner in all_players}
2 total = {field:0 for field in Map}
3
4 for ant in all_ants:
5     for field in ant.fields_in_radius(ATTACK_RADIUS + 1):
6         influence[ant.owner][field] += 1
7         total[field] += 1
8
9 def is_safe(field):
10    attacking = total[field] - influence[MY_BOT][field]
11    if attacking == 0:
12        return True
13
14    best = Inf
15    for the_field in field.fields_in_radius(ATTACK_RADIUS + 1):
16        if the_field.ant() and the_field.ant().owner != MY_BOT:
17            value = total[the_field] - influence[MY_BOT][the_field]:
18            if value < best:
19                best = value
20            if best < attacking:
21                return False
22    return True

```

Listing 4.4: Save fields evaluation

At first we need to find out how many ants are attacking each fields per ant owner and total. For this the influence and total maps are used (defined on lines 1 and 2 respectively). The influence map structure is influence[owner][field] where owner is owner of ant that can attack the field. The total is a map summing values of influence over owners. To fill the maps we find all fields in all possible attack radius after one step (line 5) for each known ant (line 4) and increase the corresponding value in our maps. At this point we can easily calculate how many ants can attack ant at any position in next turn simply by evaluating total[field] - influence[owner][field] where field is the position and owner is owner of ant at the position. This is still not enough since the ant's survival is based on how many ants is attacking the least attacked enemy ant attacking the ant.

To figure out whether the field is secure we need to get how many ants is attacking the field (line 10). If none the field is secure otherwise we need to find out how many ants is the attacker's attacking and try to find at least one that is attacking less ants that we (lines 15–19). If we find one the our ant would die on this position and thus the field is not secure (lines 20–21).



I used this method without purpose to kill enemy ant, just let ants fight those which crossed my path. Because the grow of army worked nice I let my ants to enter 1:1 situations. This nicely eliminated ants approaching my hill because of higher concentration of my ants there. This method also dramatically minimized number of deaths in 2:1 situations.

## 4.5 Better combat resolution using Minimax

I didn't focus on combat situations, but it should be noted that the winning algorithm (written by xathis) used minimax-like algorithm for combat resolution [5]. He divided ants into a groups and expanded all possible moves of that group as maximization part and all possible positions of enemies as minimization part. Evaluation function he used considered number of dead ants and also distance from enemy, to keep pushing or at least holding the battle line.

## 4.6 Symmetry detection

As the rules say, the map has to be symmetric in such way that all bots has same starting conditions. And even though the food is spawned randomly each food it is spawn in all images of that symmetry as well. It should be also noted that game starts with undiscovered map thus detecting symmetry can dramatically speed up discovering it and keeping track of food that is outside of current visible range.

If we represent symmetry as affine transformation we can generate set of possible symmetries at the start of the game and transform each newly discovered tile using each symmetry in that list and remove those that doesn't fit. Eventually we end up with a single symmetry left. This symmetry is the one that can be used to uncover additional parts of map. We can also used this symmetry to discover food that is not currently visible, but some of it's image is. Additionally the search algorithms can benefit from this, because if the search has been already run and cached on any image it can be transformed and reused.

In the end I reject this approach for several reasons. Searching for symmetry would take lot of computation time, especially at beginning of game. I found out that finding food is not a problem since there is a lot of food spawning and more important than knowing where a food is is being close to it. Knowing about food that is not close to any of my ants would take also a lot of computation time because of additional useless searches. Actually the only benefit would be faster searches later in game.

# Chapter 5

## Testing

### 5.1 Tournament

My final version of bot for tournament used A\* for food gathering and potential fields for map exploration. Unfortunately the A\* didn't have depth cut off and potential fields didn't used BFS technique described in Section 4.2 and it had problems on maze-like maps. It scored 948th place from 7897 total players.

### 5.2 Post-tournament updates

After the tournament I improved the bot and testing against the old version showed up that the new bot wins in 59.6% of games, ties in 31.2% and loses in 9% of games which seems like nice improvements. The bot uses potential fields to move ants towards the uncovered map tiles, but unlike the previous version, this time bot takes in count only tiles with path within the sensing range from new position. Reason for this change is that exploring unreachable area behind wall is useless because any food we would find wont be probable reachable. This change minimized exploring dead ends and let ants focus on more useful tasks. Another new part is that if an ant is already inside of explored and currently covered area it moves away from it neighboring ants by maximizing minimal Manhattan distance from it's neighbors. Finally the Simple Combat solution was fixed, old version of bot uses much simpler, faster but highly inaccurate solver. This solution worked pretty well while playing against bots that didn't attack on purpose. The bot was successfully avoiding positions where it would die without dealing any damage. Since the grow of army worked well I decided to allowed ants to step to position where they die and kill enemy. This decision slightly improved defence and also helped clear way to enemy hills.

### 5.3 Other bots

When the tournament started and the submitting bots was closed participants got opportunity share their bots on forum. Just several dozen of users used it and mostly only codes without description of used techniques were posted. From these posts I selected few samples

with various ranks to test my bot against them. Names in Table 5.4 are usernames users used in the official tournament and their bots can be found on the official pages<sup>1</sup>.

### 5.3.1 xathis

The winner of the tournament was Mathis Lichtenberger using nick xathis. After the tournament he also described methods he used [5]. The exploration method is very similar to mine, but instead of spreading the ants inside of covered area he sends them to borders. The borders is area that is adjacent to other bot's area. The bot's areas are generated by running BFS simultaneously from all visible bots (his and enemy's). When area collides with other one, they can be either merged in case of same owner or create border otherwise.

---

<sup>1</sup><http://aichallenge.com>

## 5.4 Results

The updated version of my bot was tested against the tournament version (listed as ficik) and several players of various rank. This testing was possible thanks to the open-sourcing act on the tournament's forums, where player have been posting sources of their bots since the submitting to tournament was closed.

Bots played all 93 tournament maps for two players and time limit was set to 300 turn. Following table show results of these duels, listed wins, loses and draws are wins, loses and draws of my bot and uses score described in Chapter 2. Since my bot was focused on food gathering and map exploring in order to grow fast in count I added ants domination and average ant difference, where domination is number of games where my player ended with more ants then enemy and the average difference is averaged difference between number my ants and enemy ants at the end of the game.

Name	Rank	wins	draws	loses	ants dominations	avg ant diff
xathis	1	3.2%	40.8%	55.9%	3.2%	-48
rossxwest	115	1.1%	50.5%	48.4%	42.0%	-6
qgazq	382	36.5%	13.9%	49.4%	66.6%	15
utoxin	726	83.8%	7.5%	8.6%	98.9%	76
Flux_w42	764	43.0%	35.5%	21.5%	74.2%	23
nmalbran	813	27.9%	40.8%	3.1%	96.8%	69
gakman	941	65.6%	26.9%	7.5%	95.7%	34
ficik	948	59.7%	31.8%	9.1%	80.6%	21

Table 5.1: Results of duels of my bot and other players from tournament

Testing bot on all maps showed that potential fields based navigation terribly suffer on maze maps especially on maps with right angle corridors with thin walls, where bot tries to navigate ant through wall. On the other hand on open maps with mostly almost convex obstacles (like these in random\_walk set) bot work very well, ants spreads across maps very fast and bot dominated in count of ants very quickly.

The combat solution also worked well against players without some sophisticated combat strategy and it would probably worked reasonably good even against xathis' bot if some technique to guard borders of of explored are was used. Ants didn't got easily killed without causing same damage to opponent. But since the ants on the edge were scattered and didn't plan moves few step further the xanthis' minimax forced them to move into deadly positions.



# Chapter 6

## Conclusion

### 6.1 Evaluation

This section lists a requirements of this thesis and describes the accomplishments.

1. The student will study the rules of the AI Challenge – Ants and the software libraries for creating players for the game.

I have described rules in Chapter 2 and used the libraries for communication with game server available at the tournament page to create the player.

2. He will review the general algorithms and techniques used in games where players control larger amount of units, mainly the potential fields and other techniques successful in the Open Real-time Strategy game.

Algorithms usable for games with large amount of units are listed and described in Chapter 3.

3. Using the available libraries, he will implement a complete player that could be submitted to the competition.

I've created and submitted player using potential fields, A\* and simple combat resolution. Games played by this player is available to review on competition website under my profile <sup>1</sup>. Sources are available online at github<sup>2</sup>.

4. He will compare the implemented player to the players successful in the 2011 AI Challenge competition.

Final ranking of the player and comparison of later version is available in Chapter 5.

---

<sup>1</sup>Profile for Ficik: <http://aichallenge.org/profile.php?user=9566>

<sup>2</sup>Repository with sources: <http://github.com/Ficik/Ants>



# Bibliography

- [1] Ai challenge. <http://aichallenge.com> [Online; accessed 18-May-2012].
- [2] Ai challenge - aichallenge/aichallenge wiki. <https://github.com/aichallenge/aichallenge/wiki/> [Online; accessed 18-May-2012].
- [3] J. Hagelbäck and S. J. Johansson. The rise of potential fields in real time strategy bots. In *AIIDE 08: Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 42–47, 2008.
- [4] J. Hagelbäck and S. J. Johansson. Using multiagent potential fields in real-time strategy games. In *Proceedings of the Seventh International Conference on Autonomous Agents and Multi-agent Systems (AAMAS)*, 2008.
- [5] M. Lichtenberger. Ai challenge 2011 (ants) post mortem by xathis.
- [6] R. G. Roie Zivan and K. Sycara. Distributed constraint optimization for large teams of mobile sensing agents. In *International Joint Conference on Web Intelligence and Intelligent Agent Technology*, pages 347–354, 2009.
- [7] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.
- [8] J. M. Vidal. *Fundamentals of Multiagent Systems*. 2007.





# CD content

▶ bin	
▼ dist	binaries of my bots
MyBot.jar	new version
MyOldBot.jar	tournament version
▶ game_logs	
▶ log	
▶ results	data generated during testing
▶ src	source codes of new version
▼ tools	package provided by aichallenge
bots	bots used for testing
mapgen	
maps	tested mapsets
sample_bots	
submission_test	
visualizer	
ants.py	
engine.py	
game.py	
playgame.py	
play_one_game.sh	
play_one_game_live.sh	
sandbox.py	
test_bot.sh	
VERSION	
visualizer.jar	
Ants-v1.0.zip	compressed source codes of tournament version
build.xml	ant build configuration
Thesis.pdf	pdf copy of this thesis
tournament	script used for testing