Analysis of Algorithms and Strategies in the Google AI Challenge 2011

Analyse der Algorithmen und Strategien in der Google AI Challenge 2011
Bachelor-Thesis von Olexandr Savchuk
März 2012

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Knowledge Engineering

Analysis of Algorithms and Strategies in the Google AI Challenge 2011
Analyse der Algorithmen und Strategien in der Google AI Challenge 2011

Vorgelegte Bachelor-Thesis von Olexandr Savchuk

1. Gutachten: Prof.Dr. Johannes Fürnkranz
2. Gutachten:

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den March 22, 2012

(Olexandr Savchuk)

Contents

1 Introduction

1.1 Artificial intelligence and computer games

Games are a hard problem that has challenged researchers in the field of artificial intelligence for a long time. Compared to many other AI problems, what makes games interesting is the combination of abstraction and high complexity. A state of a game is usually easy to represent in a computer program, and there is a limited number of actions available to the agent that all have well-defined consequences. Nevertheless, choosing the optimal action for a given game state in some games is exceptionally hard, and sometimes nearly impossible, especially considering factors like time limitations.

The first game that was played almost immediately after the advent of programmable computers was chess. A lot of modern game-playing AI techniques, including many used in this paper, originated in the early chess-playing computers. For example, a method of searching a game tree that eventually evolved to be known as the modern MIN-MAX ALGORITHM originated in the very first paper on computer chess playing published in 1950 by Claude Shannon, "Programming a Computer for Playing Chess" [Shannon, 1950].

Chess is a good example for demonstrating the complexity of a game. The average branching factor in chess is 35, and a game can often require 50 turns per side to finish, so that the full game tree contains 35^{100} or 10^{154} nodes [Russell and Norvig, 2004]. A game tree this size is impossible to store in memory of a modern-day computer, let alone traverse in limited time given for a single turn. Thus, to play chess a computer must make decisions to perform *some* moves, when an optimal move is impossible to find. Inefficiency leads to harsher penalties than in other problems, too: while a half as efficiently implemented A* search will only run twice as long to produce the same result, a chess program that plays half as efficiently in a limited time will be easily outplayed by a more efficient one [Russell and Norvig, 2004].

1.2 Google AI Challenge

The Google AI Challenge is all about creating artificial intelligence, whether you are a beginning programmer or an expert. Using one of the easy-to-use starter kits, you will create a computer program (in any language) that controls a colony of ants which fight against other colonies for domination.

(AI Challenge Homepage [Website, 2011])

The AI Challenge is an international artificial intelligence programming contest. It was started in 2010 by the Computer Science Club at the University of Waterloo. Initially the contest was for University of Waterloo students only. During the first year, the Internet search giant Google gained interest in the contest and provided sponsorship. This allowed to extend the scale of the contest dramatically, and from the second round on the AI Challenge was made available to international students and general public.

Participants in the AI Challenge write self-contained programs that play a game against one or multiple opponents. The program code gets uploaded to the contest servers, where it is compiled, tested and ultimately executed. The code can be written in a wide range of languages, for which so-called “starter kits” are provided.

Games between the programs are played on the contest servers. Game replays can be viewed on the contest website. A skill system is used to determine the ranking of all entrants based on game outcomes.

Contestants are allowed to upload any number of versions of their agents during the challenge in order to test out ideas and fine-tune their agent. At the end of the challenge, the submissions are closed, the rankings are reset, and the last uploaded versions of contestants' agents play a round of finals to determine the official rankings.

Each round of the AI Challenge uses a different game for the contestants to play. Although the first game to be used was Rock-Paper-Scissors, a widely known game, the following rounds all used original games specifically designed for the contest. Following games have been used in the past AI Challenge rounds:

- Fall 2009: Rock-Paper-Scissors
- Winter 2010: Tron Lightcycles [Website, 2010a]
- Fall 2010: Planet Wars [Website, 2010b]
- Fall 2011: Ants [Website, 2011]

The fact that the games used in the AI Challenge are unique makes it much more interesting from the research point of view. Well-known game domains used in many other contests (like chess, or rock-paper-scissors) are very deeply explored, and a very high degree of knowledge and experience is required to introduce interesting new concepts in those contexts, while still having a chance against established strategies and algorithms. By introducing a completely new domain for the participants to explore, the AI Challenge encourages them to create new approaches, experiment with different ideas and ultimately find a problem solution of their own.

Since the 2011 contest, the contest backend software is completely open-source. Many community members contributed to the AI Challenge 2011 in various ways, and the software running the contest servers is being constantly developed and improved. It must however be noted that community members who have administrative access to the contest servers are banned from participation.

1.3 Goals

The primary goal of this work is to analyze and compare the approaches utilized by the contestants of the Google AI Challenge 2011. To accomplish this, an agent is implemented according to the challenge specifications, that participates in the AI Challenge. The agent's performance is then evaluated, and a comparison is made with the approaches and performance of other competitors.

A secondary goal is to evaluate the eventual applicability of the AI Challenge as a part of the curricular learning process at the TU Darmstadt.

1.4 Structure of this paper

This paper is separated into four main chapters.

The first part (chapter 2) will introduce the reader to Ants - the game that was played in the AI Challenge 2011. It features a complete description of the game, which is necessary for understanding of certain concepts in later parts of this work. Also this part contains a brief comparison of Ants to some computer games which are deemed popular as AI application areas.

For the second part (chapter 3) a few selected ideas and algorithms were chosen and implemented in an AI Challenge agent, that then participated in the AI Challenge 2011. In this part, the motivation behind the work, the ideas that led to creation of the agent, its structure and the algorithms used are explained.

The third part (chapter 4) concerns itself with evaluating information obtained after the AI Challenge was finished. This includes an overview and detailed descriptions of some of the strongest contestants' agents and a comparison of popular approaches and algorithms.

The last part (chapter 5) is an evaluation of the agent's performance in the AI Challenge 2011. This includes observations made during the final games of the contest and their analysis, a comparison of the agent's performance to other competitors, and a list of possible improvements that didn't make it into the implementation on time.

Finally, the paper is concluded with an overview of the lessons learned from participating in a large AI programming contest, and an outlook for the possibilities regarding the future AI Challenges.

2 Problem specification: Ants

2.1 Game description

Ants is a multi-player turn-based strategy game set on a rectangular grid with wrapped edges. A game is played with a minimum of two, a maximum of ten players. Every player controls a colony of ants. The ants spawn on ant hills, of which every player has the same number. The hills and ants each occupy exactly one square of the grid. The goal of the game is to destroy ant hills belonging to other players (by moving the player's ants on top of enemy hills), while protecting the player's own hills.

On every turn any number of player's ants can be moved north, south, east or west on the grid. All players move their ants simultaneously. Ants of different players, that come within a certain distance to each other, engage in combat. Food is spawned in (semi-)random locations and times on the map (similar to the well-known game Snake); when an ant approaches a food square, that food square vanishes, and on the next turn, an ant will be spawned on one of the hills belonging to the player.

Ants is a limited information game. Players start with no knowledge of the game map, and only receive information about map areas within the field of vision of their ants, with the rest of the map being covered by "fog of war".

The ranking of players at the end of a game is determined by their point scores. A player is awarded points for destroying enemy hills and loses points for having his own hills destroyed.

The game ends, when one or more of the following conditions are met:

- Only one player's hills remain on the map.
- Only one player's ants remain on the map.
- The game has exceeded a certain number of turns.

2.1.1 Maps

A map consists of a rectangular grid of squares. Each square on the map can contain land, water, food, a hill, a live ant or dead ants. Land and hill squares can be walked on by ants, water squares cannot. Dead ants squares only exist directly after combat and can, for all intents and purposes, be treated as land squares.

Only one ant can occupy a square on a given turn. If multiple ants get moved to the same square on the same turn, they all die. If an ant is occupying a hill square, no further ant can be spawned on that hill until the occupying ant moves away or is killed.

Map edges are wrapped, which means that walking north from the top square of the map, an ant would arrive on the bottom of the map; walking off the left edge, ants arrive on the right, and so on. All pathfinding and distance calculations take this into account.

A map generator is used to generate a large number of random maps for each round of the AI Challenge. All maps are symmetric, meaning that the surroundings of every player start point are the same for all players (except mirroring or rotation), so as to equalize the chances of all contestants.

An example map is shown in figure 2.1.

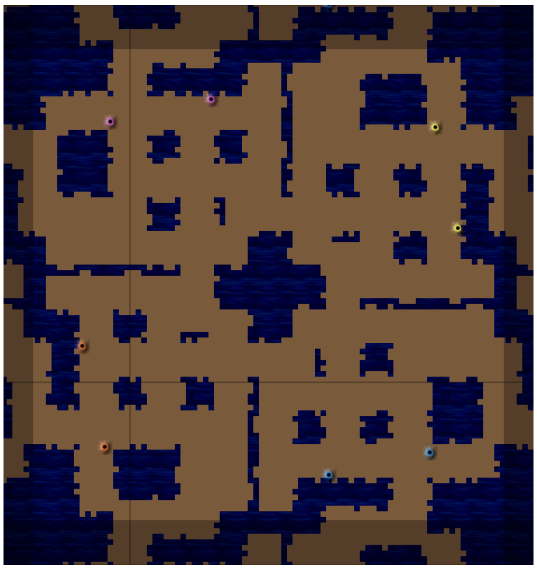


Figure 2.1: An example map for 4 players, with two hills for each player. The map symmetry is discernible. Wrapped edges are shown by repeating the map contents on all sides (darkened areas).

2.1.2 Available information

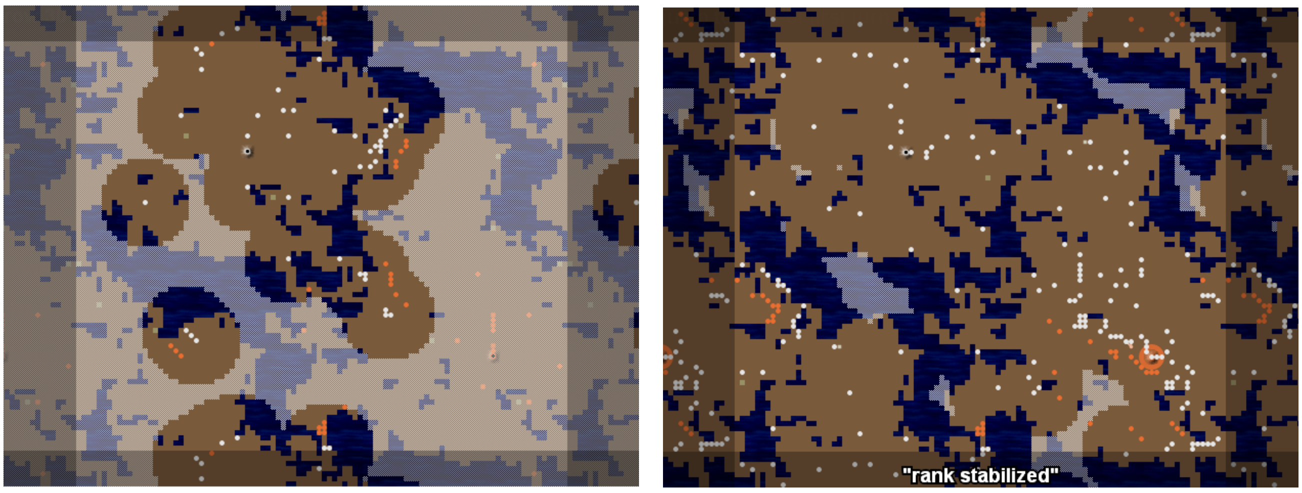
At the start of the game, the agent programs are started by the backend contest software and connected to using the specified protocol [Website, 2011, Game Specification]. The agents are then given basic information about the game: dimensions of the map, maximum turn time, as well as configuration variables (`viewradius2`, `attackradius2` and `spawnradius2` - squared values for ants' vision distance, attack distance and food gathering distance; set to 77, 5 and 1 respectively during the challenge). No information is given about the number of opponents, nor their identities. No information is given about the map other than its size.

On the beginning of each turn, the agent gets information about the current game state from the server, including currently live ants and other map objects. The ants can only "see" a certain distance around them; the rest of the map is covered by the "fog of war", a well-known concept in many other strategy games. An example game demostrating the fog-of-war mechanics is shown in figure 2.2.

The agent is provided with an explicit location of all ants which are currently within his area of vision. Every ant is identified by its coordinates and player ID. The ID is 0 for ants belonging to the player, who is currently getting the information. The first other player that a player encounters gets player ID 1, the second ID 2, and so on. Thus it is ensured that players do not know the total number of players in the game, but can still distinguish between the different opponents.

The agent is provided with explicit locations of all food squares and hills its ants can currently see. The agent also gets explicit locations of water squares, but only the first time its ants see them. On following turns, no information about already seen water squares is given to a player.

The agent is not provided with explicit information about land squares. However, since the agent knows the game's `viewradius` parameter, it can determine its field of view after getting the information about its available ants, and safely assume that every square within that field of view that was not explicitly identified as water, hill, ant or food must be a land square.



(a) Early game (b) Last turn

Figure 2.2: Two frames from an example 2-player game. Brown area is land, blue is water. White and orange dots represent ants belonging to players 1 and 2. Grey dots represent food. Frame (a) is an early game state (turn 115 of 540), frame (b) shows the final turn of the game. Fog of war is shown for player 1 (white); no information is given to that agent about areas behind the white "fog".

2.1.3 Turn phases

Every game turn begins with the game server sending the information about current the game state to all players, as described above. After that, the players have a time limit of 500ms to issue his orders, describing which ants to move and in which directions to move them. Players who do not stay within the time limit are disqualified and are not allowed to make any more moves; their ants continue to participate in the game, but do not perform any more moves.

After the server has received the orders from all players, these move orders are executed. If multiple ants (from one or more players) are moved into the same square, all of them are instantly killed, and the square becomes a land square.

Ants belonging to different players, that end up within a certain distance to each other, engage in combat with each other. For details about the combat resolution, refer to the subsection 2.1.4.

Immediately after the combat resolution, all hills currently occupied by ants not belonging to the same player as the hill are razed. The player responsible for the destruction of the hill is awarded 2 points, the hill owner loses 1 point (see 2.1.5).

After combat resolution and hill razing, spawning of new ants takes place. For every player, one ant is spawned on every hill belonging to that player. The **hive size** of that player decreases by one for every spawned ant, no further ants are spawned once the hive size reaches zero or there are no more unoccupied hills belonging to the player.

After new ants have been spawned, food is gathered. If a food square has ants from more than one player within the spawn radius of it, it disappears without further effect. If a food square has ants from a single player within the spawn radius of it, the food square disappears, and the hive size of that player is increased by one.

Finally, new food is spawned randomly (but in a symmetric fashion) across the map, that can be collected on the next turn or later.

2.1.4 Combat resolution

The combat resolution is calculated for all ants simultaneously, after the move orders are executed. The ants that are to be killed as a result of the combat resolution are marked accordingly. After the combat resolution is complete for all ants, the marked ants are removed from the game.

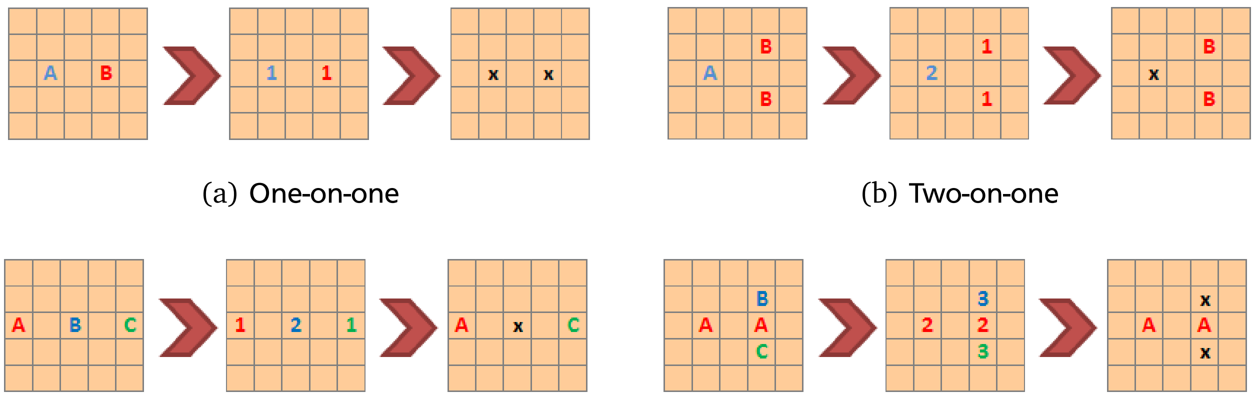
The algorithm for combat resolution for all ants in the game is described in algorithm 2.1.

Examples of combat resolution in different situations are shown in figure 2.3.

```
function CombatResolution
    for all $Ants$ do
        for all enemy ants within $AttackRadius$ of $Ant$ do
            if (enemies within $AttackRadius$ of $Ant$) >= (enemies within $AttackRadius$ of $EnemyAnt$) then
                mark $Ant$ as dead
                break enemies loop
            end if
        end for
    end for

    for all $Ants$ do
        if $Ant$ is marked as dead then
            remove $Ant$ from the game
        end if
    end for
end function
```

Algorithm 2.1: Combat resolution for all ants in the game. This algorithm describes the focus combat resolution method, that was chosen from the different methods proposed for Ants during the game development [Website, 2011].



(a) One-on-one

(b) Two-on-one

(c) "Ant sandwich"

(d) One-on-two-on-one

Figure 2.3: Combat resolution examples. In all examples, the first image indicates the players the ants belong to (A, B, C), the second image indicates the number of enemy ants within the ant's `attackradius`, and the third image indicates which ants survive the combat resolution and which are marked dead (**x**). All examples assume that $AttackRadius = \sqrt{5}$ (value used in the AI Challenge). [Website, 2011]

2.1.5 Scoring

Every player begins the game having the same number of points as he has hills on the map.

Players are awarded two points for destroying an enemy hill, and lose one point for every hill of their own that gets destroyed.

If the game ends with only one player remaining, that player is awarded bonus points. For every hill belonging to players not active anymore (crashed or timed out agent, or an agent who lost all ants), the last remaining player is awarded 2 points, and the hill owner loses 1 point.

The contest rankings take into account the player rankings at the end of a game. Score differences between players are irrelevant, position in ranking is what matters. Players with equal score are considered to be in the same position, with no further preference.

The ranking data from games is processed using a TrueSkill system [Herbrich et al., 2006], that updates skill values of all game participants. TrueSkill data of a contestant is reset when a new version is uploaded. An absolute ranking of all contestants is built based on their respective skill values.

2.2 Contestant limitations

The main limitation of the AI Challenge 2011 is that every turn must be completed in 500ms. This means that the agent must output all its commands, including the "end turn" command, within 500ms of receiving the first line of input on a turn. If an agent does not output the "end turn" command in time, it is considered "timed out" and disqualified from the game (but not from the contest), becoming unable to further control its ants and earn any more points.

All contestants submit their entries as source code in a language of their choice. The source code is compiled, tested and executed by the AI Challenge backend software running on the contest servers. No binaries can be submitted. In order for a code submission to be successful and start taking part in games against other contestants' submissions, it must pass automatic tests that verify that the program can be run without errors and communicated with using the specified protocol [Website, 2011, Game Specification].

Further limitations include only running a single thread of execution, limited process memory (1GB), no communication other than with the game server using the specified I/O protocol (specifically no network sockets), and unability to write to files persistently (all files will be reset between games). Potentially malicious practices like memory scanning or exploiting security flaws in contest software are also prohibited and punishable by contestant disqualification.

2.3 Game complexity

The naive approach to solving Ants would be a game tree of all possible moves for all ants on the field. The initial position (and thus the whole following tree) for every map is different, and a practically infinite number of different maps can be generated that are valid Ants maps. For a direct comparison, chess has one fixed starting position and is still considered a very complex game.

Further complicating the naive approach is the branching factor. In chess, the average branching factor is 35 [Russell and Norvig, 2004]. In Ants, every single ant can be moved in one of 4 directions or left standing, and in a long game, hundreds of ants can be live on the map. Assuming a total of 100 live ants in a game (a very conservative estimate), we get an astonishing raw branching factor of 5^{100}. In practice, many single moves are impossible due to terrain constraints or ants standing next to each other, or result in identical game states; but even if we reduce the average number of actions to 3 per ant instead of 5, we still get a branching factor of 3^{100}, immeasurably higher than chess or even Go (Go has an average branching factor of around 200 [Schraudolph et al., 1994], and because of this the game tree approach is usually not even attempted). Even with highly effective game tree pruning, the branching factor practically rules out the naive approach to solving Ants.

Adding to the complexity is the game not providing the players with complete information. The agents have to explore the map, locate and identify their opponents, and keep explored map areas visible to not lose information about them. Furthermore, the agents have to deal with uncertainty: the food is spawned randomly on the map and must be gathered efficiently to spawn ants, and the opponent actions must be predicted and accounted for, since they take place simultaneously with the agent's own movements.

All these factors combined result in the game of Ants showing a very high degree of complexity. Reaching good performance in the AI Challenge requires a highly sophisticated agent.

3 Agent description

Our AI Challenge 2011 agent is a program implemented in Java, that can intelligently play the game of Ants. The agent is built using a modular structure, with single modules incorporating different elements of game logic. It keeps an internal state system to model the game state during play, and employs a variety of algorithms for intelligent and quick decision-making. The key algorithm involved is a multi-purpose AdaptiveSearch (see 3.3).

3.1 Goals

The ultimate goal that the agent was supposed to achieve is play the game of Ants (as described in section 2) as intelligently as possible. That means it had to be capable of successfully destroying opponent ant hills on an unknown map, while keeping his own hills from being destroyed.

In order to reach this ultimate goal, it was broken down into components, which were each then addressed by the agent's game logic contained in its modules. The main goal breakdown was the following:

1. **Explore.** In order to destroy opponents' ant hills, the agent first needs to locate them, since their location is not known initially.
2. **Gather food.** Gathering food and spawning more ants is essential for success. Having more ants, the agent can explore quicker and counter enemy forces more effectively.
3. **Engage enemy ants in combat.** To get to the opponents' ant hills and to defend his own hills, the agent must be capable of dealing with enemy ants efficiently.
4. **Control explored territory.** Ants must be constantly present across the explored territory to keep as much of it visible as possible, to be able to see spawning food and collect it as quickly as possible, and to be able to react to opponents' ant movements.
5. **Do not exceed turn time.** Since exceeding the available time on a turn results in being disqualified from the game and losing all chances of winning points, the agent must complete all processing within the time limit.

3.2 Internal game model

Since the information presented to the agent at the beginning of a turn is limited, an internal representation of the current game state is used. This representation is formed initally after receiving game configuration values, and updated after receiving turn data at the beginning of every turn. All actions taken by the agent are also reflected in the internal state, to avoid conflicts between issued orders.

The following is a list of data that comprises the internal game model.

- **The game map.** Saved as a 2-dimensional array of tiles, the size is taken from the initial game configuration. For each tile, the following data points are available:
  - Whether the tile is visible
  - Base tile type: unexplored, land, water or hill
  - Owner of the hill, if base type is hill (player ID)
  - Whether an ant is currently occupying the tile

    - Owner of the ant, if an ant is present

- **List of the agent's hills.** Destroyed hills are removed from the list.
- **List of discovered enemy hills.** Destroyed hills are removed from the list.
- **List of the agent's available ants.** Updated at the beginning of every turn. Ants that are given orders during a turn are removed from the list.
- **List of visible enemy ants.** Updated at the beginning of every turn.
- **Turn counter.** Increased by one on every turn.
- **Start time of the current turn.** Used to determine time left on the current turn.

Information about ant, hill and food locations is saved redundantly - once in the game map and once in the instance lists. This is to simplify certain algorithms that need to iterate over these instances, but also need to identify whether instances are present at given map coordinates. Synchronisation of the data is maintained by forbidding direct access to the data structures, and using a FACADE object [Gamma et al., 1995] as an accessor instead. The implementation of the FACADE object ensures that new or updated data is synchronised in both locations.

3.3 Multi-purpose adaptive search

The multi-purpose adaptive search algorithm is utilised for finding required objects on the game map, given a set of origin squares and a list of targets or target criteria. The algorithm is utilised in many elements of the agent's playing logic. Example applications are:

- Assigning food collector ants to all visible food items
- Locating ant clusters
- Finding available ants for reinforcements in battles
- Finding closest unseen squares for exploration

The search is initialized by creating an instance of the SEARCH class. The search can be configured by utilizing the instance's setter functions (see 3.3 - Parameterization). Finally, the search can be ran once or multiple times by calling the ADAPTIVESEARCH function (algorithm 3.1).

```
function ADAPTIVESEARCH($Sources, Targets$)
    if search was configured with limited radius then
        return STATICSEARCH($Sources, Targets$)
    else if ADMISSIBLEHEURISTICAVAILABLEFOR($Targets$) then
        return EXTENDEDASTAR($Sources, Targets$)
    else
        return EXTENDEDBFS($Sources, Targets$)
    end if
end function
```

Algorithm 3.1: Multi-purpose adaptive search. Serves as a proxy method in the **Search** class, utilizes one of the available search backends depending on the configuration and target type.

Parameterization

Following parameterization options are available:

- `Seeds`: one or multiple seed squares, from which the search is started

```
function StaticSearch($Sources, Targets$)
    $Radius \leftarrow$ configuration value for maximum search radius
    $Results \leftarrow$ empty set

    if OffsetsComputedFor($Radius$) then
        $Offsets \leftarrow$ GetComputedOffsets($Radius$)
    else
        $Offsets \leftarrow$ ComputeOffsets($Radius$)
    end if

    for all $Sources$ do
        for all $Offsets$ do
            $Tile \leftarrow Source + Offset$
            if TileMeetsTargetCriteria($Tile, Targets$) then
                add $Tile$ to $Results$
            end if
        end for
    end for

    return $Results$
end function
```

Algorithm 3.2: Static search. Uses a set of coordinate offsets for a given search radius, and check all offset squares around each source for target parameters in linear time. The offsets are calculated using **lazy initialization**: the offsets for a given distance are only calculated the first time they are required, and saved for later reuse. This saves computation time for multiple searches using the same radius, e.g. in the vision module.

- `Target`: the type or coordinates of target squares
- `Radius`: the maximum distance around each seed square, at which to look for targets
- `Multi-target`: whether to look for a single closest target or all available targets
- `One per seed`: whether to limit the results to include only one target for each seed
- `Callback`: a function to be executed every time a target square is found. The callback function receives the following arguments:
  - The target square coordinates
  - The seed square where the path leading to the target originated from
  - The direction in which the path begins from the seed
  - Inverse of the direction with which the path arrives at the target

Adaptability

Since the general search algorithm was utilized on many occasions in different modules of the agent, a great variation was always present in the input data and required output. To account for that and to optimize runtimes for as many use cases as possible, different variations of the search were implemented. The concept is based on the Strategy pattern [Gamma et al., 1995].

The algorithm adapts to the available information at runtime. Depending on the options set at initialization and input data, the execution will differ as follows:

```
function EXTENDEDBFS($Sources, Targets$)
    $OpenList$ ← empty queue                                   ▷ BFS open list
    $PathSources$ ← empty map                                  ▷ for each tile, where its path originated from
    $Results$ ← empty set                                      ▷ set of all found target tiles
    $DirectionsFromSource$ ← empty map                         ▷ for each tile, the path beginning direction from respective source
    $DirectionsFromTarget$ ← empty map                         ▷ for each tile, the reversed path beginning direction from the tile
    $CompletedSources$ ← empty set                             ▷ used if the one-per-seed parameter is set

    for all $Sources$ do
        add $Source$ to $OpenList$
        $PathSources[Source]$ ← $Source$
    end for

    while $OpenList$ is not empty do
        $OpenTile$ ← $OpenList$.poll()

        $OpenTileSource$ ← $PathSources[OpenTile]$
        if $OpenTileSource$ is contained in $CompletedSources$ then
            continue while
        end if

        for all directions as $Direction$ do
            $NeighborTile$ ← GETNEIGHBORTILE($OpenTile, Direction$)
            if TILEISUNSUITABLE($NeighborTile$) or $PathSources[NeighborTile]$ is set then
                continue for
            end if

            $PathSources[NeighborTile]$ ← $OpenTileSource$
            $DirectionsFromSource[NeighborTile]$ ← $DirectionsFromSource[OpenTile]$ if set, else
$Direction$
            $DirectionsFromTarget[NeighborTile]$ ← REVERSEDIRECTION($Direction$)

            if TILEMEETSTARGETCRITERIA($NeighborTile, Targets$) then
                add $NeighborTile$ to $Results$
                if search was configured with one-per-seed parameter then
                    add $OpenTileSource$ to $CompletedSources$
                end if
            end if

            add $NeighborTile$ to $OpenList$
        end for
    end while

    return $Results, PathSources, DirectionsFromSource, DirectionsFromTarget$
end function
```

Algorithm 3.3: Extended breadth-first search. Based on a classic breadth-first search [Cormen et al., 2001, p. 531 ff.], our variation allows the use of multiple sources. All sources are added to the open set initially and are treated as normal unexpanded nodes. Search returns all found target squares, their respective sources, and two directions (path beginning direction from source and reverse path beginning direction from target).

```
function ExtendedAStar($Sources, Targets$)
    $OpenQueue$ ← empty priority queue    ▷ A* frontier, always sorted using path length and HeuristicValue
    $ExpandedNodes$ ← empty set    ▷ all nodes that were already expanded
    $PathSources$ ← empty map    ▷ for each tile, where its path originated from
    $Results$ ← empty set    ▷ set of all found target tiles
    $DirectionsFromSource$ ← empty map    ▷ for each tile, the path beginning direction from respective source
    $DirectionsFromTarget$ ← empty map    ▷ for each tile, the reversed path beginning direction from the tile
    $CompletedSources$ ← empty set    ▷ used if the one-per-seed parameter is set

    for all $Sources$ do
        add $Source$ to $OpenQueue$
        $PathSources[Source]$ ← $Source$
    end for

    while $OpenQueue$ is not empty do
        $OpenTile$ ← $OpenQueue$.poll()
        $OpenTileSource$ ← $PathSources[OpenTile]$
        if $OpenTileSource$ is contained in $CompletedSources$ then
            continue while
        end if

        if TileMeetsTargetCriteria($OpenTile, Targets$) then
            add $OpenTile$ to $Results$
            if search was configured with one-per-seed parameter then
                add $OpenTileSource$ to $CompletedSources$
            end if
        end if

        for all directions as $Direction$ do
            $NeighborTile$ ← GetNeighborTile($OpenTile, Direction$)
            if TileIsUnsuitable($NeighborTile$) or $NeighborTile$ is contained in $ExpandedNodes$ then
                continue for
            end if
            if $PathSources[NeighborTile]$ is not set or new path through $OpenTile$ is shorter then
                $PathSources[NeighborTile]$ ← $OpenTileSource$
                $DirectionsFromSource[NeighborTile]$ ← $DirectionsFromSource[OpenTile]$ if set, else
$Direction$
                $DirectionsFromTarget[NeighborTile]$ ← ReverseDirection($Direction$)
                add $NeighborTile$ to $OpenQueue$    ▷ $OpenQueue$ preserves correct sorting of nodes within
            end if
        end for
    end while

    return $Results, PathSources, DirectionsFromSource, DirectionsFromTarget$
end function

function HeuristicValue($Tile, Targets$)
    for all $Targets$ do
        $Values[Target]$ ← EuclideanDistance($Tile, Target$)
    end for
    return Min($Values$)
end function
```

Algorithm 3.4: Extended A* search. Based on a classic A* search [Russell and Norvig, 2004, p. 134 ff.], our variation allows the use of multiple sources. To maintain admissibility of the heuristic, the heuristic function is modified to return the distance to the closest target to the given square. Analogue to ExtendedBFS, all sources are initially added to the open set and treated as normal unexpanded nodes.

- If the search is limited to a certain radius surrounding each source, a StaticSearch (algorithm 3.2) will be run. This happens for example in the vision module (see 3.8.1) or in combat situation recognition (3.5.1).
- If an admissible heuristic for all potential target squares is available, an ExtendedAStar (algorithm 3.4) search will be run. This happens in certain situations in the exploration module, when ants need to be sent across long distances.
- Otherwise, an ExtendedBFS (algorithm 3.3) will be run.

Performance

General performance of the search module turned out to be satisfactory, in fact, better than one would expect from a Java implementation. A graph showing the execution times of the three search implementations depending on the distance to the target is shown in figure 3.1.

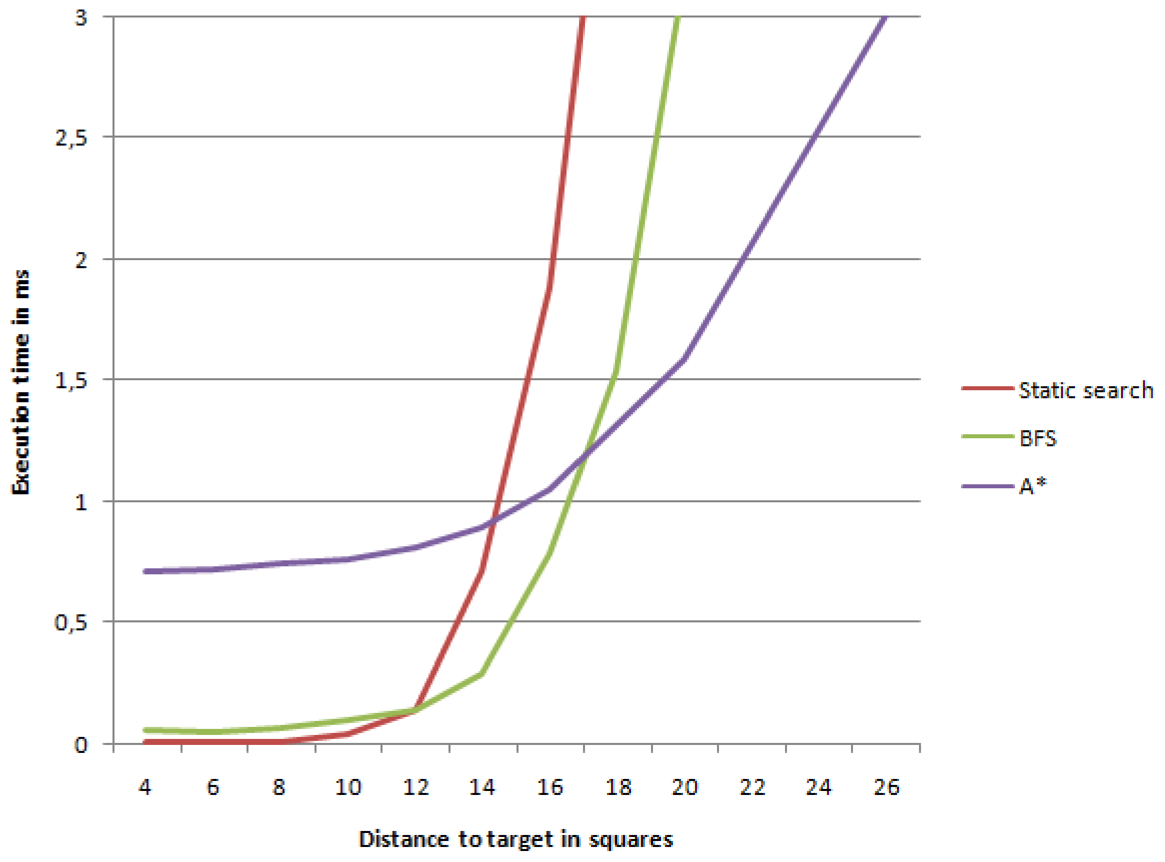


Figure 3.1: A comparison of the execution times of the search implementations within AdapriveSearch. The times were obtained by running the searches on a plain map with a set distance between a single seed and a target square. One can clearly see the advantage of the static search on low distances and the high initial overhead of the A* implementation, that is only compensated with longer search distances.

The static search was used in many situations, taking a large advantage from the faster execution time at low distances. This allowed to speed up the vision module (3.8.1) and situation recognition (3.5.1) by a large amount in comparison to the initial BFS-only approach.

All search implementations were heavily optimized to work with Java-specific data structures. By doing this, a large amount of overhead execution time was cut. This allowed the code to run at speeds comparable to test implementations in C++.

3.4 Modular structure

All game logic inside the agent is packaged into self-contained modules. These modules are responsible for single aspects of gameplay and have minimal dependencies on each other. This structure allows building and maintaining different elements of the agent separately.

Modules can be activated or deactivated independently at configuration time or at gameplay time. This allows for flexible configuration of the agent depending on what functions are required, as well as simpler maintenance and debug of single modules.

All modules are interacted with using a simple event system. At initialization time, all active modules will be instantiated and registered for their required types of events. During gameplay, certain events are fired by the game engine and modules themselves, whereupon modules that are registered for that event are notified and can act accordingly. The concept utilizes elements from the OBSERVER pattern [Gamma et al., 1995].

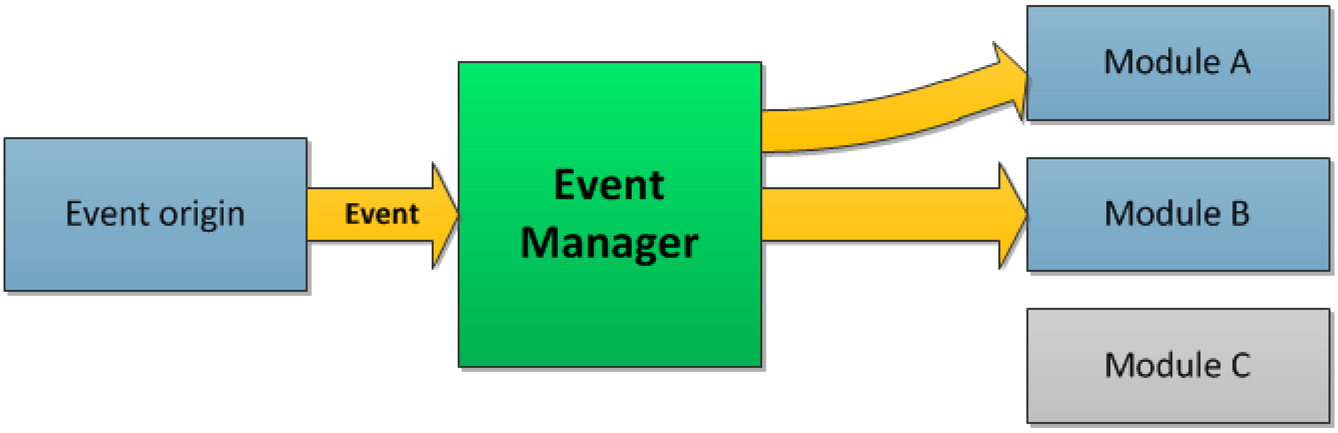


Figure 3.2: Event flow. An event is created in event origin (e.g. a module) and sent to the event manager. The event manager then propagates the event to all modules registered as listeners for the event type. Modules that are not registered for this event type are not notified.

Usage of events for passing messages between modules adds to the module system flexibility. By activating certain modules or changing their priority in the event system, it is possible to change the primary action sequence the agent undertakes during a turn without major code changes. During development, this allowed for testing multiple variations of game logic using the same code base, by reconfiguring and recombining existing modules.

3.5 Combat simulation module

The combat simulation module processes all situations where agent's ants encounter enemy ants at a certain distance. This module will try and determine the best possible move for the agent's ants in the given situation by using a state-space search [Ghallab et al., 2004], simulating agent's and enemy's possible moves and evaluating resulting situations.

Since we have established that the naive approach of building a game state tree with all possible ant movements is infeasible (see 2.3), a different approach is used here. The combat module does not simulate all ants on the map at once, instead, it first performs situation recognition and then processes single battles independent from each other. The module does not treat each ant independently like the naive approach would, but instead clusters the ants into groups and simulates coordinated group moves; this enables sophisticated combat tactics and at the same time greatly reduces computational complexity (see table 4.1).

The combat module consists of four main elements: a situation recognition block that, upon identification of a potential combat situation, finds clusters of opposing ants; a move generator, that generates possible moves to be performed by the ant groups; an evaluation function, that assigns a value to a given situation based on both sides' combat losses and heuristic values; and the main search algorithm that combines these elements to identify a best-effort move for the agent's ants in a combat situation. The element distinction is largely inspired by the similar internal structure used by the IBM Deep Blue chess computer [Campbell et al., 2002]. These elements are detailed in following sections.

3.5.1 Situation recognition

Upon encountering an enemy ant in vision range of one of the agent's ants, the module will first try to find all ants that could participate in the potential battle. The algorithm is loosely based on flood-fill, using a radius limited adaptive search in each expansion step. It works as follows:

1. Initialize $myAntsSet$ with my original ant
2. Initialize $enemyAntsSet$ with the original sighted enemy
3. Repeat following, until no new ants are added:
   a) Perform an AdaptiveSearch seeded with $myAntsSet$ looking for enemy ants, limit range to $3 * \texttt{AttackRadius}$. Add all found ants to $enemyAntsSet$.
   b) Perform an AdaptiveSearch seeded with $enemyAntsSet$ looking for my ants, limit range to $3 * \texttt{AttackRadius}$. Add all found ants to $myAntsSet$.

The two resulting ant sets are considered the current situation. The situation is considered valid, until one of the sets becomes empty. An example of situation recognition is shown in figure 3.3.

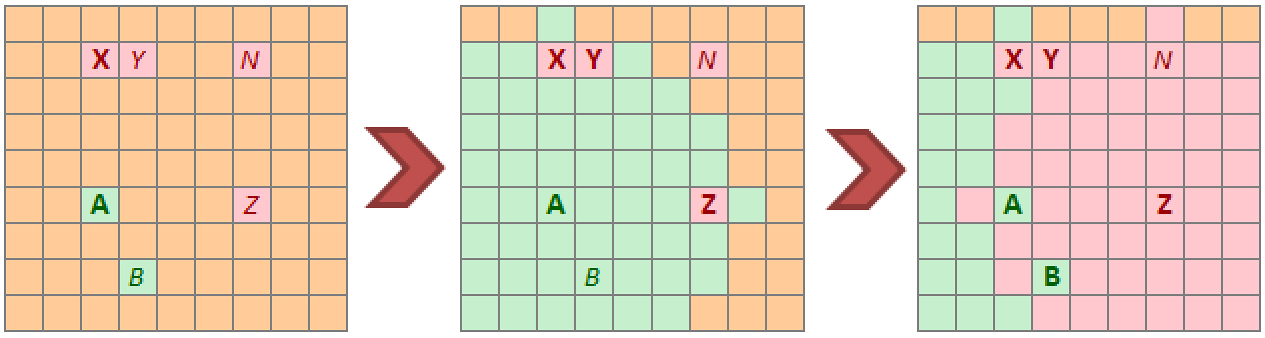


Figure 3.3: Example of the situation recognition module function. Agent's ants (A, B) are depicted in a combat situation against enemy ants (X,Y,Z,N). The situation is initialized with the ants A and X. After the first search round, ants Y and Z are added to the situation (found within search radius of A). In the next round, B is added as well (found within search radius of Z). Ant N is too far out and is not included in the situation.

3.5.2 Move generation

The move generator is the part of the combat simulation module that generates the possible moves by one of the opponent groups in a given situation. The generated moves are simulated on the given situation, generating a set of child states. These states are then evaluated by the evaluation function, and the best move is chosen. The move generator is capable of generating the following move types:

Attack In the Attack move, every ant of one side initiates a search looking for the closest enemy ant or hill, and tries to perform a step in that enemy's direction.

Hold In the Hold move, every ant will try to stay at an exact distance from enemy ants, so as to stay just outside engagement distance. The target distance will be $AttackRadius + 2$ for agent's ants (assuming the enemy side can make one move before combat resolution) and $AttackRadius + 1$ for enemy ants (since combat resolution takes part immediately after enemy ants move). The Hold move implements a safe advance move and retreat move in one, since depending on the current distance to the enemy, ants will either advance forward or retreat to reach the required distance.

Idle In the Idle move, no ants of the group will be moved.

Directional The four directional moves will simply try to move all the agent's (or enemy's) ants in one direction - north, south, east or west.

More move types were implemented and tested during development, but ultimately resulted in no visible performance improvements over this move set. Since every added move type increases the branching factor in the search, it was decided to leave out all non-essential moves and to concentrate on optimizing the basic types.

3.5.3 Heuristic state evaluation

The state evaluation is mainly based on the amount of losses on both sides in the combat resolution in this node and all the nodes along the *best child* path further down the tree. Enemy losses increase the evaluation result, while losses of the agent's ants decrease it. Bonus is given for completely eliminating all enemy ants, and a malus is subtracted if all of the agent's ants are lost. The value of enemy losses is increased, if the number of agent's ants is sufficiently higher than the number of enemy ants (overwhelming force), or when the game is nearing the final turn. The evaluation function is specified in the algorithm 3.5.

All numeric values have been manually tuned through multiple testing runs and result in appropriate behaviour. An unsuccessful attempt was made at tuning the evaluation function using genetic algorithms, see 5.3.1.

```
function EVALUATE($State$)
    $MyLossMultiplier \leftarrow 1.1$
    $EnemyLossMultiplier \leftarrow 1.0$

    if GETMYANTS($State$) > MassRatioThreshold then
        $MassRatio \leftarrow \textsc{Max}(1, (\frac{\textsc{GetMyAnts}(State)+1}{\textsc{GetEnemyAnts}(State)+1})^2)$
        $EnemyLossMultiplier \leftarrow EnemyLossMultiplier * MassRatio$    ▷ Force attacks with overwhelming force
    end if

    if GETTURNSLEFT($State$) < 50 then
        $EnemyLossMultiplier \leftarrow EnemyLossMultiplier * 1.5$    ▷ Force attacks in the game end
    end if

    $Value \leftarrow EnemyLossMultiplier * \textsc{GetEnemyLosses}(State) - MyLossMultiplier * \textsc{GetMyLosses}(State)$

    if all my ants were lost in combat then
        $Value \leftarrow Value - 0.5$
    else if all enemy ants were lost in combat then
        $Value \leftarrow Value + 0.4$
    end if

    increase $Value$ by 1 for every enemy hill destroyed
    decrease $Value$ by 5 for every agent's hill lost

    increase $Value$ by 0.5 for every food item collected by the agent's ants in the battle
    decrease $Value$ by 1 for every food item collected by enemy ants in the battle

    return $Value$
end function
```

Algorithm 3.5: Heuristic state evaluation function.

3.5.4 State-space min-max search

After the situation is identified, a search tree is built to determine the best-effort move for the agent.

In Ants, all players move their ants simultaneously. The search engine thus treats every game turn of Ants as two search plies: one ply is the agent's move, and another ply is enemy's move and turn resolution (combat and food). For this reason, the search depth is always even. See figure 3.4.

The search is initialized with a neutral node containing the current game state and the recognized situation (ant groups). After that, the move generator is used to generate the nodes for the next search ply. After the enemy move is made on every even ply, combat resolution and food collection takes place, if appropriate. Every node is then evaluated using the heuristic function.

The child node sets of every game tree node are sorted tree sets. Adding a newly generated and evaluated child node places it into the tree according to the natural ordering, which is based on the nodes' evaluation. Thus, the first tree element always contains the best move node, eliminating the need for explicit sorting.

The search depth is dynamic and limited by an upper threshold as well as the time assigned to the combat simulation.

The search algorithm is depicted in algorithm 3.6.

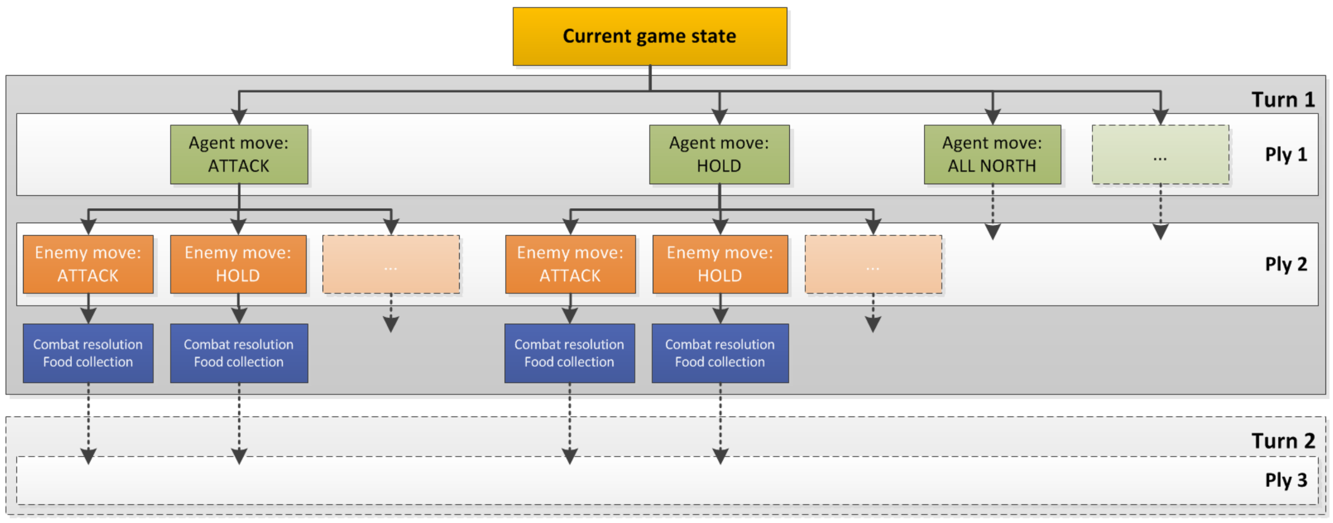


Figure 3.4: The game tree as used by the implemented min-max algorithm. Three plies are shown. The odd plies are agent moves, and even plies are enemy moves. After each even ply, a combat resolution and food collection simulation takes place.

```
function MinMax($State, Deadline, Depth = 0$)
    if not AllowExtension($Deadline, Depth$) then
        return Evaluate($State$)
    end if

    $Moves \leftarrow$ GenerateMoves($State$)
    for all $Moves$ as $Move$ do
        $ChildState \leftarrow$ PerformMove($State, Move$)
        $ChildDeadline \leftarrow \text{GetCurrentTime} + \frac{Deadline - \text{GetCurrentTime}}{\text{GetNumberOfMoves} - \text{GetMoveNumber}(Move)}$

        if IsEnemyMove($ChildState$) then
            ResolveCombatAndFoodCollection($ChildState$)    ▷ done only after enemy move ply
        end if

        MinMax($ChildState, ChildDeadline, Depth + 1$)

        add $ChildState$ to child node set of $State$    ▷ invokes Evaluate on child node and sorts the set
    end for
end function

function AllowExtension($Deadline, Depth$)
    return $Depth$ is odd or ($Depth <$ maximumdepth and $Deadline >$ GetCurrentTime+GetExtensionEstimate)
end function
```

Algorithm 3.6: State-space min-max search. The simulation is cut off at maximum depth or when not enough time is available until the deadline. Child nodes are generated by the move generator, as shown in figure 3.4. After each enemy move, combat resolution and food collection is invoked. Nodes are evaluated when being added to the child nodes tree set of each node.

3.6 Timing module

The timing module is responsible for keeping track of the limited turn time available to the agent, and informing other modules of the amount of time they have for finishing their execution. The timing module is tied in heavily with the event system, and uses the event manager to measure execution times of single modules. The measured times are then used to predict execution times of these modules on later turns, and to allocate time to modules.

The timing module reserves 10% of the allocated total turn time at the end for possible delays in communication with the game servers.

3.6.1 Timing in combat module

The combat module has by far the longest execution times of all modules, and thus it is handled separately, with some of the timing logic implemented in the combat module itself. Based on the predicted execution times of other modules, the combat module is assigned as much time for execution as possible, enabling it to maximize search depth for as many battles as possible.

When the combat module is started, it receives a timestamp indicating the maximum time it has available. The situation recognition is run first, identifying all battles that need simulating and all groups of ants participating in those battles. The time available for simulation is then split among these battles relative to the amount of ants in each battle.

The combat simulations are then executed, with the smallest battles being simulated first. The reason behind this is simple: if the smaller battles finish their simulation in less time than they were assigned, the excess time is re-assigned to the larger battles, which are simulated later.

Example: let's assume we have 220ms in total available to the combat module. Situation recognition runs for 20ms and identifies two battles in the game, with 4 ants in the first battle and 16 in the second. Of the remaining 200ms, 40ms are assigned to the first battle simulation, and 160ms to the second. If the first simulation finishes in under 40ms, the remaining time is added to the 160ms initially available to the second battle.

3.6.2 Timing in other modules

For every module except the combat module, the highest recorded execution time is tracked by the timing module. This time is used as a "safe estimate". To determine the time available to the combat module, the following formula is used:

$$T_{combat} = T_{remaining} - \sum_{m} E_m$$

where $T_{remaining}$ is the total remaining time until the end of the turn (excluding the reserve), and E_m is the estimate timing for every module m executed after the combat module.

3.7 Exploration and movement module

The exploration and movement module is responsible for directing ants around the map. Its main priorities are sending ants out to the borders of the explored area for further exploration, sending ants to areas where they are needed (ongoing battles, key points, etc.), and spreading ants across the explored area to minimize the average distance to newly spawned food and maintain a large area of vision.

The order of actions undertaken by the exploration and movement module is the following:

1. **Send out ants to unexplored area**

   This is done using an AdaptiveSearch initialized at all available ants and set to target all unexplored squares. Due to the nature of the multi-seed breadth-first search expansion on a 2-dimensional grid, only ants closest to unexplored borders successfully find paths to unexplored areas (see figure 3.5); the rest are left idle.

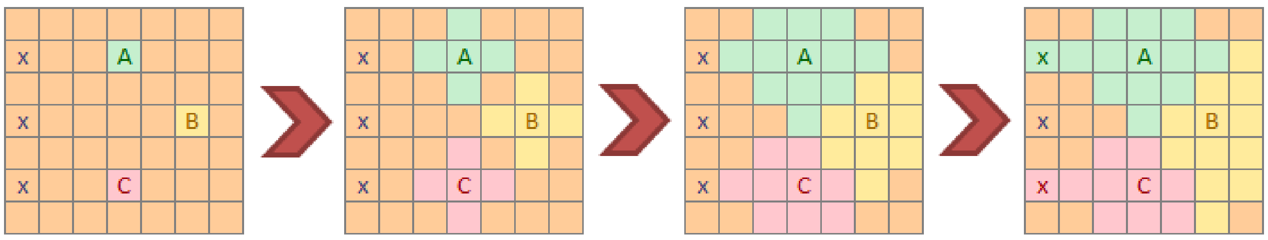


Figure 3.5: Explanation of the multi-seed BFS behaviour relevant to the exploration module function. Demonstrated is a breadth-first search seeded at squares A, B and C and targeting the squares marked with **"x"**. The expansion direction order is East-North-West-South. The colored areas indicate the initial seed from which the search reached the colored square. One can see that in the third step, the expansion from B is blocked, and that no path will be found from B to the central target.

2. **Send out ants to invisible area**

   This step uses exactly the same algorithm as the previous one, except that the targets are now invisible squares (using data provided by the Vision module, see 3.8.1). This is done to maximize vision coverage within the already explored area, to see spawning food and enemy ants.

3. **Divide ants across priority targets**

   The following priority targets are iterated over, until no more ants can be sent to any of them (either no more ants are available, no targets are present, or no paths can be found to reach the targets):

   - Unexplored area borders (redundant exploration)
   - Identified enemy ant hills
   - Visible enemy ants

   In each iteration, an AdaptiveSearch is utilized, seeded at the target set and targeting available ants.

4. **Spread out remaining ants**

In case any ants remain idle after the previous steps are completed, these ants are given orders to spread out, maximizing the distance between any two ants. Doing this serves two goals:

- Being spread out more evenly, ants can react faster to food spawning randomly between them.
- The food spawning algorithm in Ants can not spawn food on squares occupied by ants. If a food item has to be spawned on such a square, it will only appear once the ant moves away. By not letting any ants stand idle, this delay of food spawning is minimized to a single turn, allowing for faster food collection.

Another important function implemented by the exploration and movement module is that of keeping the ant hills free for spawning more ants. An ant that spawns on a hill is always moved off the hill on the same turn it is spawned, and no ants are allowed to step on the agent's hills except when it is necessary to do so in a combat situation.

The function of the exploration and movement module is best demonstrated on an artificially constructed test map used in the development of the agent, shown in figure 3.6.

3.8 Other modules

3.8.1 Vision module

Vision module is the very first module to be run at a beginning of the agent's turn. Its task is to provide other modules with information on whether given map squares are visible or covered by fog-of-war.

To accomplish this, the vision module first assigns all map squares as invisible. The squares that the agent was given explicit information about on this turn are then marked visible. The module then runs a ADAPTIVESEARCH seeded from all the agent's ants with a radius limited to the game's configured $ViewRadius$. Since the radius is limited, the search will always run a STATICSEARCH, with a minimal execution time. Every square found by the search is marked visible.

3.8.2 Food collection module

The food collection module is responsible for ants collecting as much food as possible.

The module is run during the turn computation, after the combat simulation module, to prevent putting ants in danger when sending them to collect food. Collection of food in dangerous situations is handled by the combat simulation module itself, this module only concerns itself with efficient food collection in safe situations.

When run, the following sequence of actions is performed:

1. Initialize an ADAPTIVESEARCH with all visible food items as seeds and available ants as targets
2. Set the `one-per-seed` parameter to **true** (no need to direct multiple ants to a single food square)
3. Set the `callback` function to order a move of the found ant in the direction of the food
4. Perform the search

This will result in one closest ant being sent on the way to every visible food item. If more food is visible than ants are available, the closer located food items will be preferred.

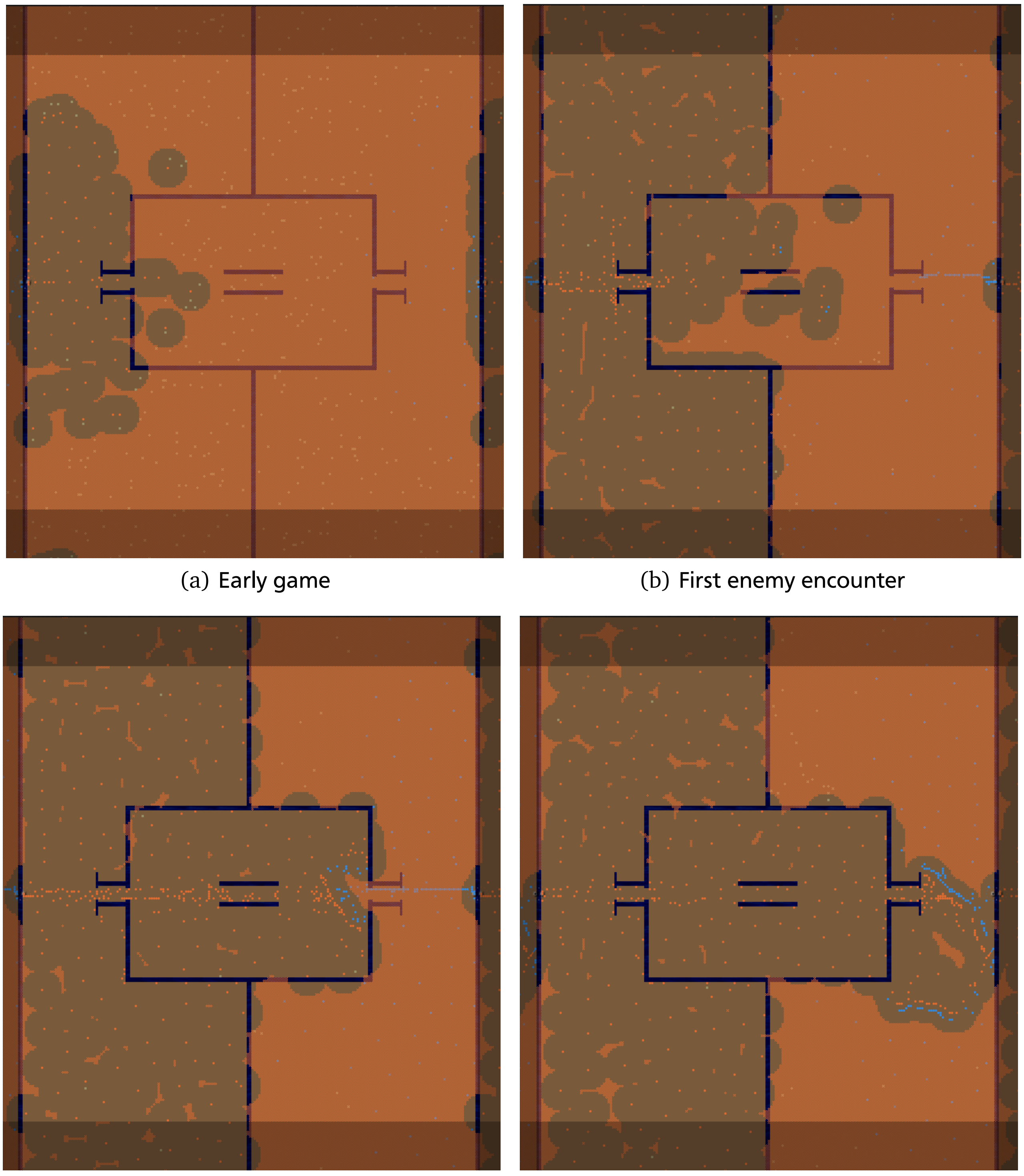

(a) Early game
(b) First enemy encounter
(c) Battle supply lines
(d) Territory control

Figure 3.6: Demonstration of the exploration and movement module function. The test map is divided in two halves by water (dark blue), with a central passage connecting both halves. In (a), one sees the even exploration around the initial ant hill (center of the left side). In (b), the left half of the map is completely explored and nearly fully visible; a battle begins in the middle, immediately all excess ants start converging on the combat location (left entrance to the central area). In (c), the ants continue flowing into combat, and the opponent is being pushed back; complete vision in the controlled map area is still fully maintained, spawned food is collected quickly. Finally, in (d) the battle is almost over, and the territory won from the opponent is also kept fully visible.

3.8.3 Hill attack and defense module

The hill attack and defense module is responsible for directing ants at the discovered opponent ant hills with the goal of their destruction, as well as for keeping enough ants within sight of agent's own ant hills and responding to enemy ants threatening them.

The attack element of this module is implemented in a very similar manner to the food collection module. An AdaptiveSearch is initialized every time an enemy hill is sighted, targeting agent's available ants and directing them to the hill. The module is only responsible for directing ants to enemy hills in "peaceful" situations; in a combat situation, the combat code will direct the ants onto the enemy hill whenever possible.

The defense system counts ants in vision radius of every agent's hill. If this number falls below a certain threshold, a search is initialized at the hill targeting available ants, resulting in additional ants being moved towards the hill until they are in vision radius. On sight of enemy ants, the agent's own ants are summoned in the same manner to defend the hill.

3.8.4 Grid module

The grid module was initially created as a means of providing vision and control over the explored territory by placing ants in pre-defined locations, allowing their vision areas to cover all explored terrain. This way, all the food that spawned in explored territory would be seen immediately and collected by the closest ant as quickly as possible.

The module utilized a static grid expanding from the initial ant starting point. To cover most area with fewest ants using the circular vision areas of each ant, a hexagonal grid was chosen, with distances between ants based on the game's `view radius` parameter, calculated so that there would be no invisible squares between the ants, while minimizing the visibility areas overlap.

While this strategy works very well on in plain areas without obstructions and requires very little computation, it becomes highly inefficient on maps with a lot of water areas. Since ants cannot be positioned on water squares, invisible areas resulted around some unreachable grid positions. Since most maps used in the contest have a lot of water squares, performance of the static grid was deemed unsatisfactory, and the exploration module was expanded instead with a different strategy for providing maximum vision.

4 Analysis of other contestants

During the contest, many different strategies and ideas for tackling the challenge were discussed by the challenge contestants in the official community and on other discussion platforms. Additionally, after the end of the AI Challenge 2011, many contestants openly published writeups of their submissions. In order to compare our implementation with others, the available sources were used to analyze the submissions of multiple contestants and compare their approaches and implementations to ours. The most interesting results of this analysis are detailed in the following section.

4.1 General approaches

The general task breakdown identified by most other contestants is largely similar to ours. Similarly, a majority of the submissions utilized similar sets of basic algorithms for the main tasks. The most widely used algorithms are, unsurprisingly:

- **Breadth first search** for finding objects on the map and paths between them
- **A* search** for quickly finding a shortest path between two points on the map
- **Min-max** for finding the best-effort move in combat situations

However, some contestants utilized different concepts entirely, with varying degrees of success. A few selected alternative approaches are detailed in the following subsections.

4.1.1 Collaborative Diffusion

An interesting approach introduced by B. Meyer [Meyer, 2011] is based on a paper "Collaborative Diffusion: Programming antiobjects" [Repenning, 2006]. This approach uses no explicit pathfinding for single ants. Instead, numeric values for different **agents** are diffused throughout the map, and ant movements are directed according to the diffusion values in squares surrounding the ant position.

> *Each square on the board has several agents or diffused values (food, explore, hill, etc). On every turn the program would loop through the map setting or diffusing all of the squares and then loop through the ants using the diffused values at the ant's square and some very basic logic to decide which way the ant should move.*
>
> *(B. Meyer's writeup [Meyer, 2011])*

The advantages of this approach are the simplicity of the program code necessary to implement it, as well as a linear algorithm complexity of $O(width * height)$ and, accordingly, low execution time and memory requirements. The agent behaviour is interesting to observe, and depending on initial diffusion values for different agents, many behavioral patterns that are forced explicitly in other submissions emerge implicitly (e.g. multiple ants exploring a corridor will naturally split up at a fork). Basic combat situation handling can also be implemented with a small extension of this approach, and was used with reasonable success by multiple contestants who didn't implement more sophisticated combat logic.

A visible disadvantage is the difficulty of fine-tuning the diffusion values of different agents. Specific behaviour more complex than what emerges from this approach naturally is also hard to implement, because of the high level of abstraction at which the decisions are made.

In the end, none of the submissions that reached top rankings in the challenge utilized this approach.

4.1.2 Contest winner: Mathis Lichtenberger (xathis)

The strategy used by the AI Challenge 2011 winner Mathis Lichtenberger (playing under the pseudonym of *xathis* [Website, 2011]) uses an efficient combination of multiple approaches, which are described below [Lichtenberger, 2011].

The exploration and territory control system utilizes numerical values assigned to all map tiles. At the beginning of the turn the value of all tiles not within the vision radius of the agent's ants is increased by 1, and the value of tiles within ants' vision (and all water tiles) is set to zero. This value signifies when a tile was last seen by the agent. A breadth-first search is then executed starting from all agent's ants up to the radius of $VisionRadius + 1$. If all squares found by the search only contain values of 0, the ant is surrounded by visible area or water, that doesn't need to be further explored. If tiles with positive values are found, the ant is moved in the direction with the highest value total, thus exploring the areas that were not seen for the longest time.

Ants that are surrounded by explored area are moved to the borders. Borders are defined to be the borders of current map exploration as well as encountered enemy ants (see below for details). Each ant is assigned a *mission*, which describes the location on the border this ant is sent to. The missions are saved persistently between turns and only deleted once the ant reaches his destination or is otherwise engaged (food collection, combat etc.).

Area and border generation is handled by an algorithm based on agglomerative clustering. Initially, every ant is assigned to its own area (cluster). A breadth-first search is initialized seeded from all ants. As the search progresses, the clusters of each ant are expanded. Clusters belonging to ants of the same players are merged as they meet. Clusters belonging to different players form a border line between them, that is then used to create missions for ants.

The concepts described above are visualized in figure 4.1.

Food collection and attacks on opponent hills are handled using breadth-first searches, in a virtually identical manner to our agent's implementation described in sections 3.8.2 and 3.8.3.

M. Lichteberger's combat algorithm is a variation of the MIN-MAX algorithm and is described in more detail in section 4.2.1.

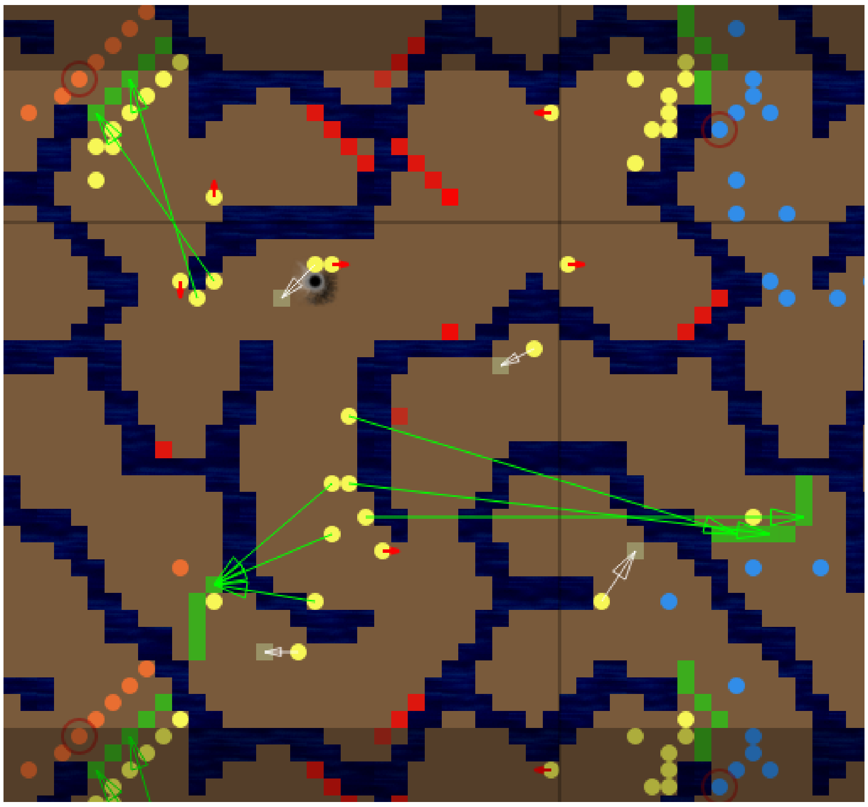


Figure 4.1: A visualization of exploration and ant movement used by M. Lichtenberger (xathis). Yellow ants are xathis' ants. Orange and blue ants are opponents. Red squares are invisible to xathis. Green squares are borders, green arrows indicate missions. Red arrows are exploration moves. White arrows indicate food collection moves. [Lichtenberger, 2011]

4.2 Combat algorithms

Without question, the most difficult part of creating an agent for the AI Challenge was implementing a combat system. This element was particularly challenging because of the high number of possible moves of an ant group, combined with the strict time limitation. All contestants who reached top rankings in the AI Challenge implemented sophisticated combat systems. In the following sections, a few selected approaches are detailed that were most widely used.

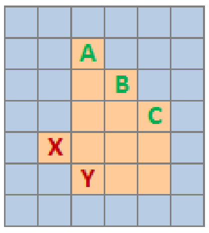
4.2.1 Min-Max variations

The most widely used algorithm for the combat system is MIN-MAX in many variations. MIN-MAX is a well-known algorithm for adversarial game tree search, and is utilized widely in many applications of AI to games such as chess. Since a straightforward application of classic MIN-MAX to Ants is barely possible due to the extremely high branching factor, some contestants (including our implementation) developed and applied variations of this algorithm.

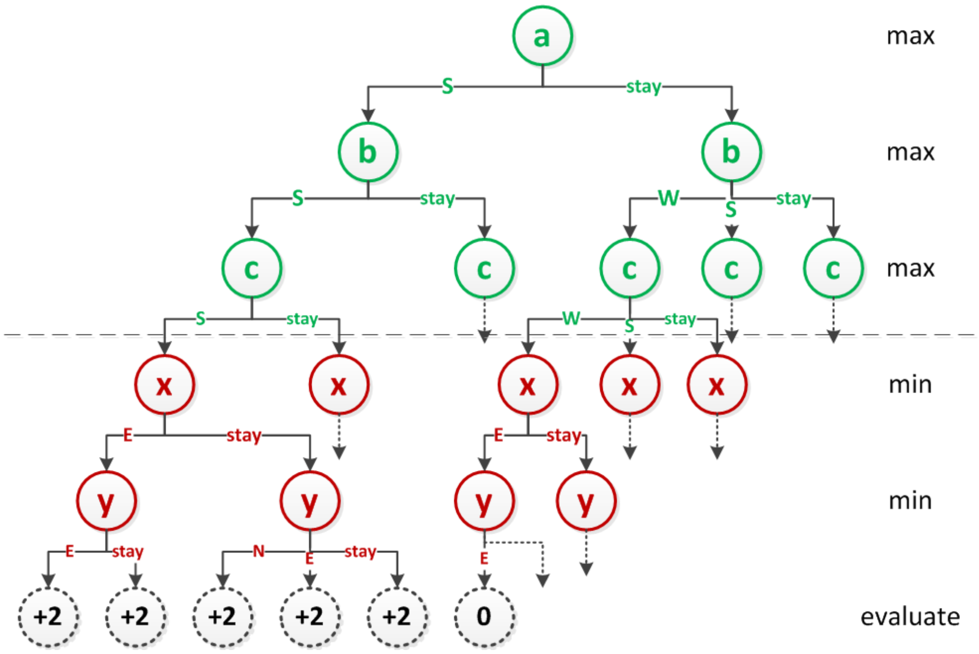
Contest winner: Mathis Lichtenberger (xathis)

One interesting approach to implementing a variation of MIN-MAX was utilized by the winner of the AI Challenge 2011, Mathis Lichtenberger from Lübeck [Lichtenberger, 2011].

Similar to our approach (see 3.5), he begins by identifying combat situations in the game and grouping ants participating in single battles together. He then builds a game tree that explores all possible moves by his ants and enemy ants for one single turn. The difference to the classic MIN-MAX [Russell and Norvig, 2004] is that tree levels with MAX and MIN nodes do not alterate, but rather multiple MAX nodes (representing moves by xathis' ants) are followed by multiple MIN nodes (opponent moves; see figure 4.2). The tree height is always equal to the number of ants involved in a single battle, and the branching factor is constant at 5 possible moves per ant, not counting optimizations. In the leaf nodes at the bottom of the tree combat simulation takes place, and the nodes are evaluated using an evaluation function much similar to the one used in our approach (see 3.5.3).



(a) Combat situation



(b) Corresponding game tree

Figure 4.2: An example game tree showing M. Lichtenberger's combat system. (a) shows the initial combat situation, and (b) the corresponding game tree generated by the min-max algorithm. Three ants belonging to the agent (A, B, C) comprise the three `max` node levels, and two enemy ants (X, Y) comprise the two `min` nodes. The final node layer are evaluation nodes, in this example simply counting the difference of ant losses on both sides.

An advantage of this approach is its completeness: all possible moves are considered, instead of just a subset of coordinated group moves. This allows the algorithm to flexibly adapt to all possible situations and not take losses due to unpredicted enemy moves.

An obvious downside is that the moves are only calculated one turn ahead. The algorithm performs very well on a tactical scale, but strategic decisions that require a situation to develop over multiple turns cannot be made by this system.

Another disadvantage is the poor scalability of the system: every ant added to the battle adds another layer of nodes to the tree, with the number of nodes increasing rapidly. The system was actually limited to battles with 7 ants per side or less, larger battles were broken down into multiple simulations, sometimes resulting in worse overall performance.

4.2.2 Alternative approaches to combat

Although the Min-Max approach to combat logic worked well for many contestants, some have chosen very different and in some cases quite innovative approaches. By far most widely used was the approach detailed below.

Influence Map by Daryl Tose (Memetix)

An alternative approach to combat was published by Daryl Tose (under the pseudonym of *Memetix*) on the AI Challenge forums [Forums, 2011] two weeks before the contest ended, and was used by multiple contestants as a result. D. Tose's own submission is ranked 10th [Website, 2011, Tose, 2011].

First, the algorithm iterates over all ants in a combat situation. It then adds a mark corresponding to that ant's player ID to every square this ant can influence after one move (all tiles within $AttackRadius + 1$). This calculates a maximum number of ants that can influence all squares relevant to the battle on the next turn. The total influence for every square (sum of marks by all player IDs on this tile) is also saved.

Next, the algorithm iterates over all tiles marked in the previous step, this time finding a number of ants an ant located in this tile would be fighting. This is calculated separately for every player, by deducting the influence by that player from the total influence on the square (leaving the influence of this player's opponents only).

Finally, the same squares from the previous step are iterated over again, this time being marked with one of three states: `SAFE`, `KILL` or `DIE`, depending on whether the agent's ant placed in this square would be fighting less enemy ants than any enemy ants around him are fighting (enemy ant dies, agent ant lives), equally many (both ants die), or more (agent ant dies, enemy ant lives; for more detailed explanation of combat resolution, see algorithm 2.1).

The rest is now simple: if the agent wants his ants to be safe and not get killed, he needs to move all of them to `SAFE` squares only. If a little risk is to be allowed (resulting in possible 1-to-1 exchanges), the `KILL` positions can be used too. The `DIE` positions are to be avoided.

A big advantage to this system is the comparable ease of computation, with a strictly linear complexity, as well as a very straightforward implementation: with some extensions on top of the described base algorithm, the combat system implemented in D. Tose's own submission is said to fit in 80 lines of code [Tose, 2011].

A further advantage of the influence map system is its good compatibility to the Collaborative Diffusion approach (4.1.1): the square states can be translated into numeric values used by the Diffusion system, and ant movement can be handled by a single function. This was noted in the discussion forum thread where the Influence Map idea was described [Forums, 2011], and led to multiple contestants utilizing a combination of these two approaches.

A large disadvantage, however, is the fact that the system performs well when used with passive, avoiding-combat behaviour, but has flaws when utilized offensively. Due to the very simple approach to computing the influence values, these are sometimes higher than is actually achievable (e.g. with ants positioned in a diagonal line, the system computes values as if multiple ants could move into the same square, which they in reality cannot do). Thus, the moves computed by this system are definitely incomplete, and in some situations where a successful attack is possible and probably desirable, agents utilizing this system would stay defensive.

4.2.3 Comparison

In table 4.1, a comparison of the combat algorithms described above is given. One can clearly see that no single algorithm is clearly supreme, and that all of them have their disadvantages that have to be either accepted or worked around with code in other agent modules.

Table 4.1: Comparison of combat algorithms

Algorithm	Branching factor	Advantages	Disadvantages
Naive game tree	$5^{number of ants}$	Complete	Infeasible to implement
MinMax (xathis)	5	Complete	Max. 7 ants per side Search depth of one turn only
MinMax (ours)	7	Variable search depth Works with very large groups	Incomplete
InfluenceMap	*linear complexity*	Execution time	Incomplete Search depth of one turn only

4.3 Other elements

Aside from general approaches and complex algorithms for combat resolution, there were many other ideas utilized by the contestants. Below a few selected ones are detailed.

4.3.1 Static formations

Static formations was a widely utilized defense strategy. Its foundations are similar to those outlined in the implemented agent's grid module description (see 3.8.4).

Agents that lacked sophisticated logic that ensured sufficient amount of ants being always present around key locations like agent's hills used a fixed number of ants positioned in a rigid formation around the location. This ensured constant vision of the location as well as a number of ants ready to respond to a potential threat (opponent ants approaching).

A disadvantage of this strategy is that being stationary, the ants used cannot be used for any other goal, like exploration or collecting food. This results in a lower number of ants active, which especially at the beginning of the game can slow the agent's progress down noticeably and provide an advantage to his opponents, who use all their ants. Also, as noted in the description of the implemented agent's exploration module (see 3.7), stationary ants can prevent food from spawning on their position.

None of the top-ranking contestants utilized static formations. Instead, sophisticated exploration and vision logic was used to reach the same goals, in many cases similar to our implementation (see 3.7).

4.3.2 Symmetry detection

Some participants utilized an untraditional approach to map exploration. As stated in the game specification (see 2.1.1), all maps in Ants are symmetric. Combining partial knowledge of the map obtained through initial exploration with this fact, locations of enemy hills and complete map knowledge can be derived, once the symmetric pattern of the map can be seen by the agent. This idea was discussed on the forum, and algorithms were developed to exploit it [Forums, 2011].

Many contestants utilized this. However, the published algorithms were quite computationally expensive, and none of the top-ranking contestants who utilized symmetry detection detailed their implementations.

5 Performance evaluation

5.1 Overall agent performance

At the end of the agent's development, the observable performance in relation to initally set goals was deemed more than satisfactory. The agent was definitely capable of playing the game of Ants intelligently, while keeping within all given limitations.

In the following list, the identified goals are listed as initially outlined in 3.1, with the corresponding observations of the agent's behaviour related to the goals.

1. **Exploration**

   At the start of the game, the agent starts exploring the area around the starting hill(s). As more ants are spawned from collected food, the exploration horizon is widened, and the ants split to explore different border areas in parallel. On encountering enemy ants, the ants take precaution and only proceed with exploration when it is safe to do so, while continuing exploration on other fronts. Given enough time, the agent consistenly completely explores 100% of accessible map areas on all types of maps.

2. **Food collection**

   Food collection is strongly prioritised by the agent. All visible food items are collected by nearby ants, as long as it's safe to do so. In combat situations, food is also collected as long as the moves necessary can be justified, and the opponents are prevented from collecting food whenever possible.

3. **Combat**

   The agent shows exceptional performance in combat. It engages the enemy ants on sight, retreats when standing against superior enemy force, gathers reinforcements from surrounding ants in the area, makes use of surrounding terrain, and aggressively pursues the enemy when at an advantage. Losses of ants in combat are minimized. The agent is also capable of making decisions in combat based on predictions of multiple turns, thanks to the variable search depth in the combat module; using this it can avoid being trapped in terrain, lead enemy ants into traps, or even implicitly apply complex tactics stretching over multiple turns.

4. **Territory control**

   Nearly all of the explored territory is kept visible until the end of the game. Only a minimal number of ants is used for this, in order to provide as many ants as possible to exploration and combat. All spawning food is collected on sight whenever possible.

5. **Keeping within time limit**

   The sophisticated timing system shows to work well. Testing was unable to reveal situations where the agent would exceed the turn time limit under any circumstances. In the final matches on the AI Challenge servers, the agent did time out in a very small number of games; this can most probably be attributed to the very high load on the servers that ran the final games, as timeouts were also observed with some of the top contestant's agents.

5.2 AI Challenge 2011 results

Submissions to the AI Challenge 2011 were closed on December 19th, 2011. The final round of matches was played until December 24th. Almost 8000 contestants took part.

The final rank of our submission was within the top **3%** of all submissions with an absolute position of **173**. It ranked **10th** among all submissions from Germany, and **52nd** among all submissions programmed in Java.

Since the AI Challenge start, the submission was re-uploaded multiple times, as the agent's implementation was iteratively refined and new features were added. An overview of the agent's absolute position in the rankings over the course of the challenge is shown in table 5.1.

Table 5.1: Agent's overall ranking over the course of the AI Challenge 2011.

Date	Version	Ranking	Total contestants	Annotations
21.10.2011	1	6	1000	First upload
24.10.2011	2	18	2000	Position drop after re-upload
27.10.2011	3	10	3000	Climbed back up
30.10.2011	3	10	4000	Position stabilized
02.11.2011	3	11	4600	
05.11.2011	3	11	5100	
08.11.2011	3	16	5600	
11.11.2011	3	22	6000	Competition stiffens
14.11.2011	3	22	6300	
17.11.2011	3	24	6500	
20.11.2011	3	27	6700	
23.11.2011	4	411	6900	Large drop after re-upload
26.11.2011	4	241	7100	Slow climb back up
29.11.2011	5	314	7200	Dropped again
02.12.2011	5	160	7300	Faster climb
05.12.2011	6	342	7500	Same pattern
08.12.2011	6	210	7550	
11.12.2011	7	150	7650	
14.12.2011	7	98	7700	
17.12.2011	7	124	7750	
19.12.2011	*final*	-	7897	**Begin of the AI Challenge finals**
24.12.2011	*final*	**173**	7897	**AI Challenge final results**

The drops in rankings after new versions were uploaded are explained by the AI Challenge implementation of the TrueSkill system: the skill values of a contestant are reset to default every time a new submission is uploaded, which moves the contestant to the back of the field. The TrueSkill system then requires multiple games to be played by the submission, until the indicated skill starts converging against its actual value [Herbrich et al., 2006].

This final performance is by far not as good as was expected during the agent's development. After analysing the results, other contestant's approaches and implementations as well as observing discussions on the AI Challenge forums, a collection of reasons explaining this outcome was compiled.

5.3 Post-challenge analysis

Detailed observation of the agent's behaviour was performed during the final AI Challenge matches. Most of the observations were consistent with the general agent performance outlined in 5.1. However, some new insights could be gained in matches against strong opponents, that were not visible in general testing or during pre-final matches.

Against most enemies, very strong combat behaviour was observed in the final games. The agent would rarely lose ants unnecessarily, and would win many engagements against groups of enemy ants. In large-scale combat, some inflexibility of the utilized group movements has started to show; for example, a large group of agent's ants surrounded by multiple enemy groups from different directions would not be able to react optimally on all fronts. The defensive behaviour in very small groups (1 or 2 ants) also sometimes led to losing ground; this was mostly directly rectified by arriving reinforcements and a counter-push.

The efficiency of the combat system seemed adequate. Even in later turns of very large games with multiple battles going on, the agent's behaviour in single battles was absolutely reasonable, which indicates that the available execution time was divided between combat simulations in an efficient manner.

The territory control module performed well, keeping vision over all explored terrain on all types of maps encountered during the final games. However, on maps with many narrow passages the system utilized more ants for this purpose than strictly necessary, compared to some of the top-ranking agents, giving some opponents a slight advantage.

A disadvantage of the dedicated combat system detached from the exploration and movement module became apparent in several games. On multiple occasions the agent would engage enemy ants in combat and continue to pursue them across the map, instead of breaking off the pursuit and going for the visible enemy hill. In most situations, agent's ants following the moving battle would destroy the hill in a short time. However, in some games the hill was left standing and cost the agent crucial points, leading to a lower resulting ranking.

Furthermore, detailed analysis of the agent's code after the challenge revealed multiple programming errors in several modules, which went undetected during development and testing. For example, due to such an error the evalutation function (3.5.3) would not always correctly identify all agent's ants being killed. This led to the agent sometimes sacrificing all of his ants in combat, when it was not necessary.

5.3.1 Elements left out

Some elements that were initially planned to be included in the agent's implementation were left out for various reasons.

Initially it was planned to fine-tune the heuristic values in the evaluation function (3.5.3) using a genetic algorithm [Russell and Norvig, 2004]. An approach for using the genetic algorithm was developed based on the work described in the paper "An evolutionary approach for the tuning of a chess evaluation function using population dynamics" [Kendall and Whitwell, 2001]. A population would consist of multiple configurations of heuristic values. Multiple games would be simulated between the population members to establish a ranking of the current population individuals. Crossover and mutation would then be used to produce a new population based on the most successful individuals of the previous one. The process would be repeated for multiple generations.

The problem that prevented this idea from being implemented was that of time. Let's assume we define our population size for a single generation to be 10 individuals (a low number for a population size). Let's further assume that in order to create an objective ranking of individuals, we only simulate 2-player games between pairs of individuals, and let every individual play a game against every other individual. A typical game of Ants can last for 500 turns and even longer. At the turn time limit of 500ms, this would mean over 4 minutes of processing time per game.

Even taking advantage of 8 available CPU threads, with 4 games of 2 players each being run in parallel, this still means one minute for every game. With 10 individuals having to play each other, computing the game outcomes for a single generation would take 45 minutes. In the paper used as a basis for this approach, genetic learning proved successful after 45 generations [Kendall and Whitwell, 2001], which would mean that the process would have to run for 34 hours to produce results. This was not possible.

Furthermore, this calculation assumes we only let each pair of individuals play against each other on one map. The results of this would not be very representative, as there were many maps played in the AI Challenge with great variations between them.

5.3.2 Possible improvements

Based on the post-challenge analysis of our agent's performance and comparison to other competitors' implementations, a number of possible improvements to the agent were identified, that would have led to a better overall performance.

Better testing procedures would constitute one such improvement. A test suite would be designed, incorporating testing of the vital functions of the agent, as well as some very specific test cases of combat and exploration logic. Such a test suite might have enabled a better discovery of programming errors and led to a lower number of them being present in the final submission.

The genetic optimization approach could be used in a modified form. Since the main optimization would affect only the evaluation function used in the combat logic, it would be possible to only simulate prepared combat situations instead of whole games, drastically cutting the required simulation time per pair of individuals in a population. Unfortunately, this possibility was not discovered before the submission deadline.

6 Conclusion

6.1 Summary

A number of algorithms were designed to provide a working solution for the game of Ants, played in the Google AI Challenge 2011. The algorithms were based on common AI algorithms utilized in other game-playing AIs, such as chess computers. An agent was implemented incorporating these algorithms. The agent has successfully participated in the AI Challenge, showing intelligent play as originally intended and finishing within the top 3% of all contestants.

The results of the AI Challenge were evaluated. Approaches and algorithms chosen by other participants were analyzed, and in many cases found to be largely similar to the ones chosen for the implemented agent. Comparisons between the performance of algorithms of the implemented agent and those of other competitors' agents were made. Interesting ideas and algorithms implemented by the competitors were discussed.

An in-depth analysis of the agent's performance in the AI Challenge was conducted. Advantages and disadvantages of the chosen approach were discussed using knowledge obtained from the analysis of other competitors' submissions. Elements were identified that could be improved to possibly boost the overall performance.

6.2 Future prospects

Direct reuse of the work completed for this paper is limited. The game of Ants was played in the AI Challenge 2011 only. Upcoming AI Challenges will utilize new games designed specifically for the Challenge, rendering the currently implemented agent unapplicable.

However, due to the abstract nature of Ants, many elements implemented in the agent can be potentially reused for creating an artificial intelligence in other strategy games, possibly including future AI Challenge games. Utilized general-purpose algorithms like the AdaptiveSearch can be used in different contexts with minimal adaptation necessary. The modular structure and time-keeping module can both be used in other contexts as well.

An interesting future prospect is a potential application of the AI Challenge as a teaching instrument. The unique conditions of the AI Challenge make it a very challenging enviroment. To devise a complete working solution for an unexplored problem domain in limited time takes a lot of skill and time investment. It requires application of well-developed and tested techniques together with a creative approach. In the case of Ants, the fact that no single optimal solution was discovered by the AI Challenge contestants makes it even more interesting, since varying approaches could be investigated to reach similar levels of performance. The results of work being immediately visible helps improve the learning experience greatly.

A presentation of this thesis was conducted in the Oberseminar Knowledge Engineering on January 25th, 2012. This presentation has awakened interest in the AI Challenge among students and teaching staff. As of March 2012, a group is being devised at the TU Darmstadt with the intention of letting students participate in the future AI Challenges as a curricular activity.

Bibliography

C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.

Stuart J. Russell and Peter Norvig. *Künstliche Intelligenz - ein moderner Ansatz (2. Aufl.)*. Pearson Studium, 2004. ISBN 978-3-8273-7089-1.

AI Challenge Website. Ai challenge official website, fall 2011: Ants. Website, 2011. Available online at `http://ants.aichallenge.org`; visited on March 15th 2012.

AI Challenge Website. Google ai challenge official website, winter 2010: Tron. Website, 2010a. Available online at `http://tron.aichallenge.org`; visited on March 15th 2012.

AI Challenge Website. Google ai challenge official website, fall 2010: Planet wars. Website, 2010b. Available online at `http://planetwars.aichallenge.org`; visited on March 15th 2012.

Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill$^{\text{tm}}$: A bayesian skill rating system. In *Neural Information Processing Systems*, pages 569–576, 2006.

N.N. Schraudolph, P. Dayan, and T.J. Sejnowski. Temporal difference learning of position evaluation in the game of go. *Advances in Neural Information Processing Systems*, pages 817–817, 1994.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001. ISBN 0-262-03293-7; 0-07-013151-1.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

M. Campbell, A.J. Hoane, and F. Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

Benjamin Meyer. Benjamin meyer's blog: Using collaborative diffusion rather than path finding for the google ai ant challenge. Website, 2011. Available online at `http://benjamin-meyer.blogspot.de/2011/11/using-collaborative-diffusion-rather.html`; visited on March 19th 2012.

A. Repenning. Collaborative diffusion: programming antiobjects. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 574–585. ACM, 2006.

Mathis Lichtenberger. Ai challenge 2011 (ants) post-mortem by xathis. Website, 2011. Available online at `http://xathis.com/posts/ai-challenge-2011-ants.html`; visited on March 19th 2012.

AI Challenge Forums. Ai challenge forums - strategy. Website, 2011. Available online at `http://forums.aichallenge.org/viewforum.php?f=24`; visited on March 20th 2012.

Daryl Tose. Ai challenge 2011 writeup by memetix. Website, 2011. Available online at `http://www.blackmoor.org.uk/AI.htm`; visited on March 19th 2012.

G. Kendall and G. Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 2, pages 995–1002. IEEE, 2001.

List of Figures

List of Tables

List of Algorithms