Requirements

Contents

1 User Requirements **3**

2 Functional Requirements **4**

2.1 Game 4

2.2 Ant 7

2.3 World 11

2.4 GUI 13

2.5 Parsing Specifications 14

2.5.1 Brain 14

2.5.2 World 15

3 Non-Functional Requirements **16**

3.1 Product 16

3.1.1 Reliability 16

3.1.2 Usability 16

3.1.3 Performance 16

3.1.4 Security 17

3.1.5 Portability 17

3.1.6 Resources 17
3.1.7 Deployment 17
3.2 Organisation 17
3.3 External 18

4 Domain Requirements **19**

1 User Requirements

1. The program should check if an ant-brain supplied by a player is syntactically well-formed.
2. The program should check if a given description of an ant world is syntactically well-formed and meets the requirements for ant worlds used in tournaments.
3. The program should be able visualise a given ant world.
4. The program should be able to generate random but well-formed ant worlds.
5. The program should allow two players to play: i.e. enable two players to upload their ant-brains and choose an ant-world, and then run the game in the ant world, taking statistics and determining the winner of the game.
6. The program should allow users to play tournaments, where an arbitrary number of players can upload ant-brains, who are all paired up to play against each other. The overall tournament winner is the ant brain that wins the most individual games.
7. The development team should also produce a novel, proof-of-concept ant-brain.
8. The user can upload brains into an ant world with two ant colonies, one for each player.
9. The program should also support uploading custom ant world from files.
10. An ant world should simulate the behaviour of both kinds of ants, using the ant-brains supplied by the players.
11. The two species of ants are placed in a random world containing two anthills, some food sources, and several obstacles.
12. Ants must explore the world, find food and bring it back to their anthill.
13. Ants can communicate or leave trails by means of chemical markers.
14. Each species of ants can sense (with limited capabilities), but not modify, the markers of the other species.
15. Ants can also attack the ants of the other species by surrounding them
16. Ants that die as a result of an attack become food.
17. If an ant is adjacent to 5 (or 6) ants of the other species then it dies. However, if an ant is in a cell in the very corner of the grid or along one of the edges it can only have at most 4 ants adjacent to it - therefore it cannot be killed.
18. The match is won by the species with the most food in its anthill at the end of 300,000 rounds.
19. A cell can hold an unlimited amount of food during the course of the simulation.
20. In a tournament, each team will play every other team twice on each of the contest worlds. Once as red and once as black. So for 23 teams, each team will be pitted up against 22 other teams.
21. There must be at least one empty cell between adjacent non-food items.

2 Functional Requirements

2.1 Game

Game/1

Specification	Details
Requirement	Simulate a two-player game.
Description	Using two ant-brains and an ant-world, carry out a simulation of the game based on the ant finite-state machines and their environment.
Inputs	Two different colonies (consisting of a colour, a team name and an ant-brain) and an ant-world.
Source	The GUI event loop.
Outputs	The final result of the game (which team won/lost). The final state of the world.
Destination	The GUI event loop.
Actions	For each turn (maximum 300,000), the ants on each cell shall be iterated over. Each ant will perform an action based on their ant-brain (specified as a finite-state machine and graph). The state of a cell (and possibly other ants) will be updated based on the action of the ant. The winner of the game is determined by the colony with the most food in their anthill (that is, the number of food particles in their anthill's cells). Food carried by ants is ignored in this score counting.
Notes	-

Game/2

Specification	Details
Requirement	Produce statistics for an ant game.

Description	By keeping track of the state of the world at all times (or having access to the history of the game), for each team the following statistics will be computed: • Amount of food in anthill • Number of foe's ants killed • Number of team's ants left • Number of movements made • Number of markings left Further statistics can be added at the discretion of the programmers where convenient.
Inputs	The state of an ant-game at a particular turn (accumulative) or the full history of an ant-game at the end of the game.
Source	An ant game.
Outputs	The statistics as specified above.
Destination	The GUI statistics display screen.
Actions	Given the state of the game at a particular turn, all relevant statistics counters shall be updated. At the end of the game, the statistics shall be calculated and output.
Notes	-

Game/3

Specification	Details
Requirement	Simulate a tournament.
Description	Plays ant games between multiple users, each with their own ant-brain specification.
Inputs	A list of teams (specified by a name and an ant-brain), and an arbitrary (but at least 1) number of ant-worlds.
Source	The GUI.
Outputs	The results of all matches, as well as the overall winner (the team with the highest number of wins).
Destination	The GUI tournament statistics display screen.

Actions	Iterates over all unique combinations `(team-1, team-2, world)` and simulates a game with that 3-tuple, with the first team playing as red and the second team playing as black. Keeps track of all wins, losses and statistics for each team. After all simulations, (which are *not* visualised to the user as per `Game/1.3` but rather the statistics are displayed) the overall winner is returned. The final results shall be calculated as follows: a team shall earn 2 points for a win, and 1 point for a draw. The score shall be tallied at the end, with the team having the highest score being the winner. If there is a draw, the tournament shall be repeated with the highest-scoring half of teams.
Notes	Iterating over all unique 3-tuples as described above will mean that every team will play every other team on every world as both red and black. The number of worlds used for a tournament shall be specified by the user when interacting with the GUI tournament setup view.

Game/4

Specification	Details
Requirement	Parse an ant brain file.
Description	Checks if a user's custom ant brain (expected to have been provided in a file on disk) is valid according to the ant brain specification. If so, constructs an in-memory programmatic representation of the brain.
Inputs	A user-specified path to the ant-brain file.
Source	A file in-memory that has been read from disk.
Outputs	A programmatic representation of the ant brain.
Destination	The GUI event loop.
Actions	Iterates through each line of the file, tokenising the line and validating each token. Builds a graph of ant-brain instructions which form a finite-state machine with transitions between instructions dependent on the conditions specified in the line.
Notes	The specification of an ant-brain file is further below: see `Parsing Specifications/Ant Brain`.

Miscellaneous

Identifier	Requirement	Rationale
Game/Misc.1	At the start of a game, all ants shall face east.	User-requested.
Game/Misc.2	Ant's brains shall be initialised to state 0.	User requested.
Game/Misc.3	Each anthill cell shall be initially populated with a single ant of the anthill cell's colour.	User-requested.
Game/Misc.4	Ants shall be allocated identifiers in increasing numerical order, in top-to-bottom left-to-right order based on their position in the ant world.	User-requested.

2.2 Ant

Ant/1

Specification	Details
Requirement	Sense a cell.
Description	Performs a sense operation on a particular cell of a particular condition (cell and condition specified by the ant-brain instruction).
Inputs	Ant-brain instruction context, access to the cell in question.
Source	The game controller.
Outputs	A boolean result that informs the game controller which state the ant-brain has transitioned to (either the success state or failure-state, as specified in the original ant-brain file).
Destination	The game control loop.
Actions	The ant's current instruction stores information about the direction to sense in and the condition to sense for. The direction is one of `here`, `ahead`, `left-ahead` or `right-ahead`. The ant queries the world to determine the cell in the relevant direction, and checks the condition on that cell. If the condition holds for that cell, it transitions to the `success` state. If not, it transitions to the `fail`-state.
Notes	The predicates available to an ant for sense operations are `friend` (whether the given cell contains an ant of the same team), `foe` (whether the given cell contains an ant of the opposing team), `friend_with_food` (as with `friend`, but also carrying food), `foe_with_food` (as with `foe`, but also carrying food), `rock` (whether the cell type is rocky), `marker n` (whether the cell is marked with a marker, represented by an integer `n`, of the same team), `foe_marker` (whether the cell has marked by a foe, with *any* integer marker identifier), `home` (whether the cell is part of the anthill of that ants team), and `foe_home` (whether the cell is part of the opponent's anthill).

Ant/2

Specification	Details
Requirement	Mark a cell.
Description	Performs a mark operation on the current cell, placing a chemical marker as specified by the current ant-brain instruction.
Inputs	The world context, the ant-brain instruction context.
Source	The game controller.
Outputs	No output required.
Destination	The game control loop.
Actions	Tells the world to mark the ant's current cell with a particular integer. Transitions to the next state.

Notes	The marker must be in the range `0..5`.

Ant/3

Specification	Details
Requirement	Unmark a cell.
Description	Performs an unmark operation on the current cell, removing a chemical marker as specified by the current ant-brain instruction.
Inputs	The world context, the ant-brain instruction context.
Source	The game controller.
Outputs	No output required.
Destination	The game control loop.
Actions	Tells the world to unmark the ant's current cell, removing the marker with a particular identifier. Transitions to the next state.
Notes	The marker must be in the range `0..5`.

Ant/4

Specification	Details
Requirement	Pick up food.
Description	Picks up food from the current cell, transitioning to one of two ant-brain states depending on whether the pick-up operation was successful.
Inputs	The world context, the ant-brain instruction context.
Source	The game controller.
Outputs	A boolean result that informs the game controller which state the ant-brain has transitioned to (either the success state or failure-state, as specified in the original ant-brain file).
Destination	The game control loop.
Actions	Queries the world to see if the current cell has food. If not, transitions to the fail-state. Otherwise ets the internal state of the ant to `has-food = true`, as per requirement `Ant/Misc.4`. Removes a food particle from the current cell. Transitions to the success state.
Notes	-

Ant/5

Specification	Details
Requirement	Drop food.
Description	Drops food into the current cell, transitioning to the next state as specified in the ant-brain finite state machine.
Inputs	The world context, the ant-brain instruction context.

Source	The game controller.
Outputs	No output required.
Destination	The game control loop.
Actions	Gets the current cell from the world context and adds a food particle to it. Transitions to the next state.
Notes	If the ant does not have food, no action on the cell shall be taken, and the ant shall transition to the next state.

Ant/6

Specification	Details
Requirement	Turn.
Description	Causes the internal direction attribute of the ant (as specified in `Ant/Misc.7`) to change based on the current turn instruction's specification.
Inputs	The world context, the ant-brain instruction context.
Source	The game controller.
Outputs	No output required.
Destination	The game control loop.
Actions	The direction specified by the instruction can be one of `left` or `right`. This shall cause the direction of the ant to 'rotate' - i.e. `left` will cause the ant to rotate one edge anticlockwise in its hexagonal cell, and vice-versa for `right`. The ant will then transition into the next state.
Notes	-

Ant/7

Specification	Details
Requirement	Move.
Description	Causes the ant to move forward based on its direction into an adjacent cell (if possible).
Inputs	The world context, the ant-brain instruction context.
Source	The game controller.
Outputs	A boolean result that informs the game controller which state the ant-brain has transitioned to (either the success state or failure-state, as specified in the original ant-brain file).
Destination	The game control loop.

Actions	Based on the ant's current direction and the ant's current cell, the world can be queried for the cell in front of the ant. If the cell is free (not rocky and does not contain another ant), the ant shall remove itself from the current cell and add itself to the relevant adjacent cell. It shall then update its own `resting` attribute to indicate to the game controller that it must now rest for 14 moves. Then it will transition to the success state. If not free, the ant shall transition to the fail-state.
Notes	Resting is not *handled* here, rather it is handled by the game controller which will check the resting attribute and not call any instruction-step methods on the ant unless not resting.

Ant/8

Specification	Details
Requirement	Flip.
Description	Causes the ant to transition into one of two states depending on the result of a pseudo-random generated integer.
Inputs	The world context, the ant-brain instruction context.
Source	The game controller.
Outputs	A boolean result that informs the game controller which state the ant-brain has transitioned to (either state-1 or state-2, as specified in the original ant-brain file).
Destination	The game control loop.
Actions	Using the system's random number generator, the ant shall produce a random integer between `0..(n-1)`, where `n` is the integer specified by the flip instruction. If `0`, the ant shall transition to state-1. Otherwise, it shall transition to state-2.
Notes	-

Miscellaneous

Identifier	Requirement	Rationale
Ant/Misc.1	Each ant shall have an associated unique integer identifier.	This identifier shall determine the order in which the ant moves as compared to other ants (with ants moving in ascending order of identifier).
Ant/Misc.2	Each ant shall have an associated colour, either `red` or `black`.	The colour shall identify the ant's team and also make it clear to the user in world visualisation.

Ant/Misc.3	Each ant shall use the finite-state machine representation of its brain to store its current state.	This state is queried by the game controller to decide how the ant interacts with the world.
Ant/Misc.4	Each ant shall hold a `boolean` field that tracks whether the ant is holding food.	Accessible to the game controller since it performs different actions depending on whether an ant carries food or not.
Ant/Misc.5	Each ant shall hold an `integer` field that tracks how many turns it is resting for.	When an ant rests, it is not allowed to move. This field shall be decremented by the game loop, and if `0` then the ant is able to move.
Ant/Misc.6	Ants shall die if surrounded by `5` ants of the opposing team.	User-specified requirement. Note: this means that ants on the edge of the game-world or next to certain rocky structures are not able to be killed, since they do not have `5` free cells surrounding them.
Ant/Misc.7	The ant shall track its own direction.	The direction of the ant shall be used by the game controller for operations such as moving the ant forward or sensing.

2.3 World

World/1

Specification	Details
Requirement	Parse ant world file.
Description	Checks if a user's custom ant world (expected to have been provided in a file on disk) is valid according to the ant world specification. If so, constructs an in-memory programmatic representation of the world.
Inputs	A user-specified path to the ant-world file.
Source	A file in-memory that has been read from disk.
Outputs	A programmatic representation of the ant world.
Destination	The GUI event loop.
Actions	Iterates through each line of the file, tokenising the line and validating each token. Intepreting each token as a cell and building a hexagonal programmatic structure containing cells with properties specified in the file.
Notes	-

World/2

Specification	Details
Requirement	Generate random contest worlds.
Description	Creates random contest worlds that conform to the requirements for a tournament, as described in the *Actions* section below.
Inputs	None.
Source	The tournament setup, or alternatively the GUI. That is, this has two different operations - either the user requests a custom ant world (to write to file), or the contest world is randomly generated as a tournament is starting.
Outputs	A programmatic representation of an ant world.
Destination	The GUI event loop (or tournament setup).
Actions	Pseudo-randomly generates an ant world of 150x150 cells. This ant world shall have rocky edge cells. It shall have two hexagonal ant hills of edge length 7, for red and black, randomly located. It shall have 14 randomly located rocks. It shall have 11 randomly located blobs of food, where a blob of food is a 5x5 rectangle. Each food blob shall be randomly oriented and each cell in the ant blob shall have 5 food particles. All separate components of the world (ant-hills, food-blobs, rocks) shall be separated by at least one clear cell.
Notes	A component for converting a programmatic representation of an ant world to a textual representation should also be developed.

Miscellaneous

Identifier	Requirement	Rationale
World/Misc.1	The ant world programmatic structure shall present an interface that appears as if the world is comprised of hexagonal cell (that is, each non-edge cell should be surrounded by 6 adjacent cells).	As specified in the user requirements, each cell is a hexagon.
World/Misc.2	Each cell shall have an associated type, which shall be one of `rocky`, `clear`, `red-ant-hill` or `black-ant-hill`.	As per user requirements, `rocky` cells are inaccessible to ants, and `ant-hill` cells belong to their associated teams and are where ants are initially spawned.
World/Misc.4	Ant hills cannot be adjacent to one another.	Direct user requirement, reasoning unclear.
World/Misc.5	Each non-rocky cell can contain `0-1` ants.	Direct user requirement. Ants cannot 'walk over' each other.

World/Misc.6	Each clear cell can contain `0+` food particles.	Food cannot logically be negative. Note: as specified by the ant-world cell specification, initially a cell can contain only `0-9` food particles. However, as food is dropped this number can increase above the initial limit unboundedly.
World/Misc.7	Each clear cell can contain any non-negative number of chemical markers for each team.	Ants can mark the same cell with multiple markers, but cannot modify their opponents.

2.4 GUI

GUI/1

Specification	Details
Requirement	Visualise ant world.
Description	Uses a Swing GUI to show users a representation of the ant world in a game at a particular turn.
Inputs	An ant world containing information about cells, including their location, their type, and what they contain (e.g. ants, food).
Source	A currently-running ant game.
Outputs	An image (or otherwise) showing the state of the game in a human-readable fashion, displayed using a Swing interface.
Destination	The game loop.
Actions	Loops through all cells in the game and draws them to screen, with an indication of the type of cell and what it contains specified by a colour (colours not yet specified, will be at the discretion of the programmers and testers).
Notes	-

Miscellaneous

Identifier	Requirement	Rationale
GUI/Misc.1	The GUI shall display the entire world, including all world content.	Allows viewing of the entire state of the game in a user-friendly way.
GUI/Misc.2	The game view interface shall scale proportionally to world dimensions `(x, y)`.	Scaling means the entire world state is always visible to users.

GUI/Misc.3	There will be a user-customisable option to display the state of the game every `n` frames.	Makes the game play-speed personalisable to the user, to avoid simulation being too short or too long.

2.5 Parsing Specifications

2.5.1 Brain

This is the specification for an ant-brain file. These are provided by the user, but development of the brain parser shall interpret the file contents as conforming to the following (taken directly from the client requirements):

- Each line in the file represents one state. The first line is state 0, the second line state 1, and so on.
- The file may contain at most 10000 lines.
- Each line shall consists of a sequence of whitespace-separated tokens, followed (optionally) by a comment beginning with a semicolon and extending to the end of the line.
- Tokens are either keywords or integers.
- Keywords are case-insensitive.
- The possible instruction tokens are:
  - `Sense`
  - `Mark`
  - `Unmark`
  - `PickUp`
  - `Drop`
  - `Turn`
  - `Move`
  - `Flip`
- The tokens for sensing directions are:
  - `Here`
  - `Ahead`
  - `LeftAhead`
  - `RightAhead`
- The tokens for conditions are:
  - `Friend`
  - `Foe`
  - `FriendWithFood`
  - `FoeWithFood`
  - `Food`
  - `Rock`

  - `Marker i`, where `i` is an integer from `0-5`
  - `FoeMarker`
  - `Home`
  - `FoeHome`
- The tokens for turn are:
  - `Left`
  - `Right`

2.5.2 World

This is the specification for an ant-world file. These are both provided by the user and generated randomly, so development of the world parser and contest world generator shall conform to the following (taken directly from the client requirements):

- The first line of the file contains a single integer representing the size of the world in the `x` dimension.
- The second line of the file contains a single integer representing the size of the world in the `y` dimension.
- The rest of the file consists of `y` lines, each containing `x` one-character cell specifiers, separated by spaces (even lines also contain a leading space before the first cell specifier). The top-left cell specifier corresponds to position `(0, 0)`. **Note**: although generated worlds shall conform to this specification, the parser shall ignore leading and extra whitespace.
- The cell specifiers are:
  - `#`: rocky cell
  - `.`: clear cell (containing nothing interesting)
  - `+`: red anthill cell
  - `-`: black anthill cell
  - `1-9`: clear cell containing the given number of food particles

3 Non-Functional Requirements

3.1 Product

3.1.1 Reliability

1. The software should crash no more than 1 time in 1000 when parsing user files, even if the files are corrupt or not well-formed.
2. The software should crash no more than 1 time in 1000 when simulating a game or tournament.
3. The GUI should be stable and responsive to user input at least 99% of the time, including when a game is running. This applies only to systems with more than 400MB of free RAM.
4. The software shall parse user files of up to 10,000 lines each without crashing more than 1 time in 1000.
5. Visualisations of games and tournaments should proceed with no visible defects (glitches) at least 99 times out of 100.
6. If the program crashes, a log containing the stack trace of the error and any other pertinent details shall be written to the program's working directory, titled "ant-game-crash-[current unix time].log".

3.1.2 Usability

1. The software should have a minimal learning curve for new users. That is, the GUI should have intuitive and clear controls that step the user(s) through each stage of the game without being required to follow external instructions (not including understanding the rules of the game and the development of a user's ant-brain file).
2. All user interfaces shall be developed using the JavaFX API.
3. The software shall present a help menu, accessible at all times in the program, which upon clicking displays a popup user interface containing the searchable user documentation for the program.

3.1.3 Performance

1. Response time of the system when interacting with any component of the GUI should be negligible before the user is presented with the interaction's result or with a loading screen.
2. The GUI should not hang (that is, be unresponsive to user interaction) for more than one second after any user interaction.
3. Parsing user files (ant-brain, ant-world) should take no longer than 5 seconds.
4. The two-player game simulation should take no longer than 30 seconds for 300,000 rounds, not including artificially introduced delays when displaying the game to the user.

3.1.4 Security

1. The software shall not require internet or network access to run, and will have access to the full feature-set of the program offline without use of any network connections.
2. The system shall not purposefully access or alter any existing user or system files, with the exception of ant-brain and ant-world files chosen with explicit permission from the user (via selection from a file-chooser interface).

3.1.5 Portability

1. The software shall be cross-platform across commonly used personal desktop operating systems (Windows 7+, Mac OS X 10.8.3+, Linux) that support Java 8.0.

3.1.6 Resources

1. The system should not store more than 100MB of non-volatile data, not including the user's ant-brain and ant-world files.
2. The software shall use no more than 100MB of RAM, with the exception of a maximum of 400MB of RAM while executing/displaying a game.

3.1.7 Deployment

1. A build of the program shall be delivered in executable JAR format, to allow the user to run the program without accessing the command line and without any other external configuration by the user.

3.2 Organisation

1. All development and testing shall be undertaken in either the Netbeans IDE or IntelliJ IDE.
2. Team communication relevant to the product-related aspects of the project (planning, development, testing) shall be recorded for future reference:
    1. Facebook chat - message archive
    2. Slack chat - message archive
    3. In-person - meeting minutes
3. JUnit 4.12 shall be used as the testing library across all development and testing systems.
4. Gradle 2.5+ shall be used as the build automaton system across all development and testing systems.

5. All project-related documents, source code, tests and builds shall be kept on the team's GitHub repository, with new versions being committed and pushed to the repository on a regular basis and as soon as a major change/refactoring occurs.
6. The software's source code shall be documented extensively, defined as follows:
    1. The Javadoc comment structure (`http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`) shall be used to document every field, method and class.
    2. The structure of Javadoc documentation shall follow the Oracle style guide (`http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`).
    3. Single-line comments shall be used inside methods to make the purpose of lines/blocks of code clear, when deemed appropriate by the coder(s).
    4. For fragments of incomplete code which are pushed to the project's GitHub repository or left to complete later, a single line comment starting with "//TODO:" shall be used to briefly describe the task to complete.
7. A copy of all project documents and source code shall be stored independently on each team member's personal computer, regularly pulled from the central team GitHub repository. In addition, an up-to-date backup of the GitHub repository shall be kept in one or more of the team's personal Dropbox repositories.
8. The Jacoco code coverage analysis tool shall be used by the testing/validation team to ensure maximum code coverage. The final submitted system shall have 100% code coverage.

3.3 External

1. The software shall be delivered by Thursday 5 May at 4:00PM, by online submission to Study Direct.
2. All libraries and external material used in the project shall be documented and credited to the original author(s), as per the University of Sussex's plagiarism (`http://www.sussex.ac.uk/s3/?id=35`) policies.

4 Domain Requirements

There are no additional domain requirements for this project. The application domain of the system is assumed to be extremely similar to the development and testing domain: as per the non-functional product requirements, this software is designed to run on personal operating systems compatible with Java 8.

All current requirements have been classified as functional or non-functional.