Ants

Enric Rodríguez

April 30, 2024

1 Game rules

In this game each player controls a colony of ants. At each round of a match, players accumulate as many points to their score as the number of ants in their colony at the end of the round. The winner of the game is the player who, at the end of the match, has the highest score.

The game is played on a rectangular board of dimensions *BOARD_ROWS* $\times$ *BOARD_COLS*. Cells of the board can be of *water* or *soil*. Each ant occupies a single cell, and there cannot be two ants on the same cell at the same time. Ants are afraid of water and hence can only occupy cells of soil.

In colonies there are several categories of ants: *queens*, *soldiers* and *workers*. Ants have a finite life, which is decremented by one after each round. When the life of an ant is exhausted, the ant dies. Initially, the life of a queen is *QUEEN_LIFE* rounds, the life of a soldier is *SOLDIER_LIFE* rounds, and the life of a worker is *WORKER_LIFE* rounds.

Every colony has one queen. If the queen of a colony dies, then another ant of the colony (soldier or worker), if there is any, is chosen randomly and becomes the new queen. The life of this new queen is *QUEEN_LIFE* rounds.

A match consists of *NUM_ROUNDS* rounds. In a round, ants can move to one of the adjacent cells of soil within the board, following a horitzontal or a vertical direction (but not in diagonal). Queens can only move in rounds which are multiples of *QUEEN_PERIOD*. For example, if *QUEEN_PERIOD* is 2, queens can only move at rounds 0, 2, 4, 6, etc. Soldiers and workers can move at any round.

When an ant tries to move to a cell which is already occupied by another ant (be it of the same colony or not), the former attacks the latter. The result of the fight is determined as follows:

- If the two ants are of the same category, both die.
- Queens kill soldiers and workers.

- Soldiers kill workers.

If the attacking ant survives, the movement is performed.

Independently of the presence of an ant, some soil cells may also contain a bonus of food of different kinds: *bread*, *seeds* or *leafs*. When a queen moves to a cell with food, it eats the bonus. On the other hand, when a worker moves to a cell with food, it can take up the food, carry it for some time and leave it some cells away. Food can only be left at the current position of the worker. A worker can only carry one bonus at a time, and can only leave it on cells that do not contain any bonus already. When a worker that carries food dies (because its life is exhausted, or as a result of a fight), the bonus gets destroyed. Soldiers cannot eat or carry food.

A queen may lay an egg (of a soldier or of a worker) in an adjacent (horitzontally or vertically) soil cell. At the end of the round the egg hatches and a new ant of the colony is born. If the newborn is a soldier, its life is *SOLDIER_LIFE* rounds, and if it is a worker, it is *WORKER_LIFE* rounds. However, if at the moment the egg hatches the cell is already occupied by another ant (be it of the same colony or not), the new ant dies automatically.

A queen needs three kinds of nutrients in its reserve to produce eggs: *carbohydrates*, *proteins* and *lipids*. Depending on whether the queen decides to lay an egg of a soldier or of a worker, the required quantity of these nutrients is different. The constants that define these quantities are summarized in the following table:

	Soldier	**Worker**
Carbohydrate	*SOLDIER_CARBO*	*WORKER_CARBO*
Protein	*SOLDIER_PROTE*	*WORKER_PROTE*
Lipid	*SOLDIER_LIPID*	*WORKER_LIPID*

When a queen eats, the nutrients of the food accumulate to its reserve. The nutrients of each kind of food are determined by the constants in the following table:

	Bread	**Seed**	**Leaf**
Carbohydrate	*BREAD_CARBO*	*SEED_CARBO*	*LEAF_CARBO*
Protein	*BREAD_PROTE*	*SEED_PROTE*	*LEAF_PROTE*
Lipid	*BREAD_LIPID*	*SEED_LIPID*	*LEAF_LIPID*

When the match begins, a colony consists of its queen, *NUM_INI_SOLDIERS* soldiers and *NUM_INI_WORKERS* workers placed on soil cells at one of the four corners of the board (player 0 at the top-left one, player 1 at the top-right one, player 2 at the bottom-right one, and player 3 at the bottom-left one). Initially the queen has no nutrients to lay eggs, that is, its reserve is empty. Similarly, when a soldier or a worker becomes a queen it has no nutrients either. There is an exception, though: if the ant was a worker carrying food, the bonus

is consumed and its nutrients accumulate to the reserve of the new queen; the same happens if the cell it occupies contains food. If the two situations take place, both bonuses are eaten.

For each of the four quadrants of the board there is a rectangle of *BONUS_ROWS* $\times$ *BONUS_COLS* cells included in the quadrant on which bonuses of bread may appear. There are also analogous rectangles for seeds and leafs. For each of these rectangles, every *BONUS_PERIOD* rounds (starting from round 0) a soil cell of that rectangle without bonuses and without an ant is chosen randomly and a new bonus appears. If there is no such cell, then no new food bonus is created. The exact location of these rectangles is determined randomly and is unknown to the players.

Each ant has a natural number that uniquely identifies it. In a round, players use these identifiers to command their ants to perform actions, at most one for each ant:

- moving to an adjacent soil cell;
- taking up food;
- leaving food;
- laying an egg of a soldier or a worker.

Only the first action commanded to an ant is selected for execution. The rest of the commands for that ant are ignored. Moreover, any player program that tries to give more than 1000 commands in the same round is aborted.

At the end of every round, all selected commands are ordered randomly and executed following this order. If a command cannot be executed (e.g., the commanded ant has died as a consequence of previously executed commands), the command is skipped. After executing all commands, life counters are decremented, and ants whose life has been exhausted die. Then eggs laid by queens hatch, also in random order (even if the queen that laid the egg has died, e.g., due to a fight). Then, for each colony that no longer has a queen, a new queen is chosen randomly among the remaining soldiers and workers, if there are any, as explained above. Then food bonuses (bread, seeds and leafs) are generated if the current number of rounds is a multiple of *BONUS_PERIOD*. Finally for each player the number of ants of the colony is added to the score.

1.1 Game parameters

A game is defined by a board and the set of parameters of Figure 1.

Unless there is a force majeure event, the indicated values of the parameters are the ones that will be used in all matches of the game.

Parameter	Value	Description
NUM_PLAYERS	4	Number of players in the game.
NUM_ROUNDS	250	Number of rounds a match lasts.
BOARD_ROWS	25	Number of rows of the board.
BOARD_COLS	25	Number of columns of the board.
QUEEN_PERIOD	2	Queens are allowed to move every *QUEEN_PERIOD* rounds, starting from round 0.
SOLDIER_CARBO	3	Units of carbohydrates needed for an egg of a soldier.
SOLDIER_PROTE	3	Units of proteins needed for an egg of a soldier.
SOLDIER_LIPID	3	Units of lipids needed for an egg of a soldier.
WORKER_CARBO	1	Units of carbohydrates needed for an egg of a worker.
WORKER_PROTE	1	Units of proteins needed for an egg of a worker.
WORKER_LIPID	1	Units of lipids needed for an egg of a worker.
BREAD_CARBO	2	Units of carbohydrates contained in a bonus of bread.
BREAD_PROTE	0	Units of proteins contained in a bonus of bread.
BREAD_LIPID	1	Units of lipids contained in a bonus of bread.
SEED_CARBO	0	Units of carbohydrates contained in a bonus of seed.
SEED_PROTE	1	Units of proteins contained in a bonus of seed.
SEED_LIPID	2	Units of lipids contained in a bonus of seed.
LEAF_CARBO	1	Units of carbohydrates contained in a bonus of leaf.
LEAF_PROTE	2	Units of proteins contained in a bonus of leaf.
LEAF_LIPID	0	Units of lipids contained in a bonus of leaf.
NUM_INI_SOLDIERS	3	Number of initial soldiers.
NUM_INI_WORKERS	11	Number of initial workers.
BONUS_ROWS	3	Number of rows of a rectangle where food appears.
BONUS_COLS	3	Number of columns of a rectangle where food appears.
BONUS_PERIOD	25	Food bonuses appear every *BONUS_PERIOD* rounds, starting from round 0.
WORKER_LIFE	75	Rounds after which a worker dies.
SOLDIER_LIFE	150	Rounds after which a soldier dies.
QUEEN_LIFE	300	Rounds after which a queen dies.

Figure 1: Game parameters.

2 The Viewer

In Figure 2 a screenshot with all the elements of the game is shown.

- On the top left corner there are buttons that allow one to play/pause the match, go to the beginning or the end of the match, toggle on or off the animation mode, or get a help window with further ways of controlling how the match is played. On the top right corner there is a button for closing the viewer. The current round is also shown on the top of the viewer. A horizontal slide bar visually shows in which point of the match this round is.
- In the column on the left, every player appears with the corresponding name and color. The reserve of the queen is displayed below (if a player has no queen, then all values are 0). In the matches played in Jutge.org, it is also shown the percentage of CPU time that has been consumed so far (if exhausted, this is indicated with an 'out').
- In the column on the right, the score of each player is displayed in order, being the highest score the one on the top.
- The cells of soil are painted in light brown, whereas the cells of water are painted in light blue.
- Queens, soldiers and workers are represented as in Figure 3.
- Bonuses are represented as in Figure 4.
- Workers that are carrying a bonus have a circle around them of different color according to the kind of bonus: orange for bread, green for leafs and grey for seeds. In the screenshot, there are four workers carrying bread, another one carrying leaf, and yet another one carrying seed.

3 Programming

The first thing you should do is to download the source code. This source code includes a C++ program that runs the matches and also an HTML viewer to watch them in a nice animated format. Also, a "Null" player and a "Demo" player are provided to make it easier to start coding your own player.

3.1 Running your first match

Here we will explain how to run the game under Linux, but a similar procedure should work as well under Windows, Mac, FreeBSD, OpenSolaris, ... The only requirements on your system are `g++`, `make` and a modern browser like Mozilla Firefox or Google Chrome.

To run your first match, follow the next steps:

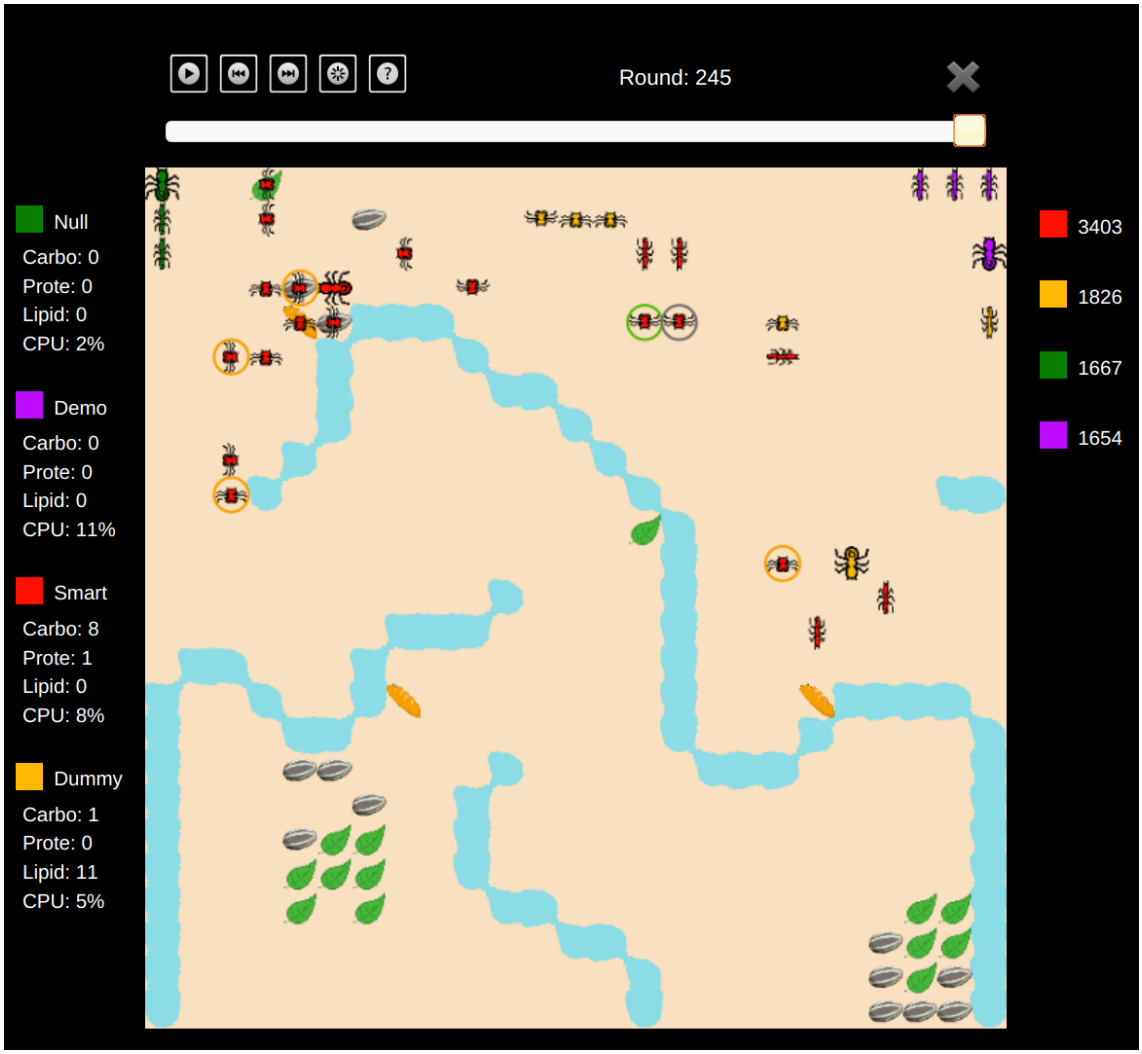

Figure 2: Screenshot of the game.

Figure 3: Representation of a queen, a soldier and a worker.

Figure 4: Representation of a bread, leaf and seed bonus.

1. Open a console and `cd` to the directory where you extracted the source code.
2. If, for example, you are using a 64-bit Linux version, run:

   ```
   cp AIDummy.o.Linux64 AIDummy.o
   ```

   If you use any other architecture, choose the right objects you will find in the directory.
3. Run

   ```
   make all
   ```

   to build the game and all the players. Note that `Makefile` identifies any file matching `AI*.cc` as a player.
4. This creates an executable file called `Game`. This executable allows you to run a match using a command like:

   ```
   ./Game Demo Demo Demo Demo -s 30 -i default.cnf -o default.out
   ```

   In this case, this runs a match with random seed 30 where four instances of the player "Demo" play with the parameters defined in `default.cnf` (the default parameters). The output of this match is redirected to the file `default.out`.
5. To watch a match, open the viewer file `viewer.html` in the directory `Viewer` with your browser and load the file `default.out`.

   Note: To get a larger view of the match, put your browser into full screen mode (press key F11).

Use

```
./Game --help
```

to see the list of parameters that you can use. Particularly useful is

```
./Game --list
```

to show all the registered player names.

If needed, remember that you can run

```
make clean
```

to delete the executable and object files and start over the build.

3.2 Adding your player

To create a new player with, say, name `MyPlayer`, copy `AINull.cc` (an empty player that is provided as a template) to a new file `AIMyPlayer.cc`. Then, edit the new file and change the

#define PLAYER_NAME Null

line to

#define PLAYER_NAME MyPlayer

The name you choose for your player must be unique, non-offensive and less than 12 letters long. It will be used to define a new class *PLAYER_NAME*, which will be referred to below as your player class. The name will be shown as well when viewing the matches and on the website.

Now you can start implementing the method *play* () . This method will be called every round and is where your player should decide what to do, and do it. Of course, you can define auxiliary methods and variables inside your player class, but the entry point of your code will always be this *play* () method.

From your player class you can also call functions to access the board state, as defined in the *State* class in `State.hh`, and to command your units, as defined in the *Action* class in `Action.hh`. These functions are made available to your code using multiple inheritance. The documentation on the available functions can be found in the aforementioned header files. You can also examine the code of the "Demo" player in `AIDemo.cc` as an example of how to use these functions. Finally, it may be worth as well to have a look at the files `Structs.hh` for useful data structures, `Random.hh` for random generation utilities, `Settings.hh` for looking up the game settings and `Player.hh` for the *me*() method.

Note that you should not modify the *factory* () method from your player class, nor the last line that adds your player to the list of registered players.

3.3 Playing against the "Dummy" player

In order to try your strategy against the Dummy player, we provide you with its object file. Hence, you do not have access to its source code but can add it as a player and compete against it.

As we have already mentioned, in order to add the Dummy player to the list of registered users, you must copy the file corresponding to your architecture to `AIDummy.o`. For example:

```
cp AIDummy.o.Linux64 AIDummy.o
```

Remind that object files contain binary instructions for a concrete architecture. This is why we cannot provide a single file.

Pro tip: ask you friend their **object** files (never source code!!!) and add them to your `Makefile`.

3.4 Restrictions when submitting your player

Once you think your player is strong enough to enter the competition, you should submit it to the Jutge.org website (`https://www.jutge.org`). Since it will run in a secure environment to prevent cheating, some restrictions apply to your code:

- All your source code must be in a single file (like `AIMyPlayer.cc`).
- You cannot use global variables (instead, use attributes in your class).
- You are only allowed to use standard libraries like `iostream`, `vector`, `map`, `set`, `queue`, `algorithm`, `cmath`, ... In many cases, you don't even need to include the corresponding library.
- You cannot open files nor do any other system calls (threads, forks, ...).
- Your CPU time and memory usage will be limited, while they are not in your local environment when executing with `./Game`. The timelimit is one second for the whole match. If you exceed this limit (or you execution aborts), your player will be frozen and will not admit any more commands.
- Your program should not write to **cout** nor read from **cin**. You can write debug information to **cerr** (but remember that doing so in the code you upload to the server can waste part of your limited CPU time).
- Any submission to Jutge.org must be an honest attempt to play the game. Any attempt to cheat in any way will be severely penalized.
- Once you have submitted a player to Jutge that has defeated the Dummy player, you can send more submissions but you will have to change the player name. That is, once a player has defeated Dummy, his name is blocked and cannot be reused.

4 Tips

- **DO NOT GIVE OR ASK YOUR CODE TO/FROM ANYBODY.** Not even an old version. Not even to your best friend. Not even from students of previous years. We will use plagiarism detectors to compare pairwise all submissions and also with submissions from previous editions. However, you can share the compiled `.o` files.

  Any detected plagiarism will result in an **overall grade of 0** in the course (not only in the Game) of all involved students. Additional disciplinary measures might also be taken. If student A and B are involved, measures will be applied to both of them, independently of who created the original code. No exceptions will be made under any circumstances.

- Before competing with your classmates, focus on qualifying and defeating the "Dummy" player.

- Read only the headers of the classes in the provided source code. Do not worry about the private parts nor the implementation.

- Start with simple strategies, easy to code and debug, since this is exactly what you will need at the beginning.

- Define basic auxiliary methods, and make sure they work properly.

- Try to keep your code clean. Then it will be easier to change it and to add new strategies.

- As usual, compile and test your code often. It is *much* easier to trace a bug when you only have changed few lines of code.

- Use `cerr`s to output debug information and add `assert`s to make sure the code is doing what it should do. Remember to remove (or comment out) the `cerr`s before uploading your code to Jutge.org. Otherwise, your submission will be killed.

- When debugging a player, remove the `cerr`s you may have in the other players' code, to make sure you only see the messages you want.

- By using commands like `grep` in Linux you can filter the output that `Game` produces.

- Switch on the `DEBUG` option in the Makefile, which will allow you to get useful backtraces when your program crashes. There is also a `PROFILE` option you can use for code optimization.

- If using `cerr` is not enough to debug your code, learn how to use `valgrind`, `gdb`, `ddd` or any other debugging tool. They are quite useful!

- You can analyze the files that the program `Game` produces as output, which describe how the board evolves after each round.

- Keep a copy of the old versions of your player. When a new version is ready, make it fight against the previous ones to measure the improvement.
- When a player is submitted to the Jutge.org server or during the competition, matches are run with different random seeds. So when training locally, run matches with different random seeds too (with the `-s` option of `Game`).
- Make sure your program is fast enough: the CPU time you are allowed to use is rather short.
- Try to figure out the strategies of your competitors by watching matches. This way you can try to defend against them or even improve them in your own player.
- Do not wait till the last minute to submit your player. When there are lots of submissions at the same time, it will take longer for the server to run the matches, and it might be too late!
- You can submit new versions of your program at any time.
- And again: Keep your code simple, build often, test often. Or you will regret.