Dominator

Omer Giménez Jordi Petit Enric Rodríguez
Salvador Roura Albert Vaca

November 21, 2016

1 Game rules

In this game, four players have control over an army of farmers, knights and witches on a 37×37 board.

The goal of the game is to "farm" as many cells as possible, converting them to the team color. Initially, the cells have no color. Every round, the number of cells of each color is added to the corresponding score. The winner of the game is the player with the highest score after the last round.

Each player has a quadrant where their units are initially born, and where they are reborn when captured (killed) from other teams. Player 0 has the top-left quadrant, player 1 the bottom-left quadrant, player 2 the bottom-right quadrant, and player 3 the top-right quadrant. At the beginning of the game, each player gets 20 farmers, 10 knights and 2 witches on its quadrant.

The game lasts 200 rounds. Every round, every player can move each own unit at most once. Farmers and witches can only move horizontally and vertically. Knights can also move diagonally, and attack rival units.

The boards have walls, that is, obstacles that units cannot visit nor traverse. The top row, bottom row, lef-most column, and right-most column only have walls. Trying to move a unit into a wall is an invalid move.

Initially, farmers have health 100, and knights have health 200. Deliberately not moving a unit (or moving it in the `None` direction) will increase its health by 30 units. A unit cannot have more health than its initial amount. Performing an invalid move results in that unit not moving, but does not regenerate health.

Witches are immortal. Witches can move to any adjacent empty cell. Any cell at Manhattan distance at most two from a witch (that is, at most two horizontal or

vertical steps away from a witch, or diagonally adjacent to a witch) is haunted, even if there is a wall in between. Any farmer or knight in a haunted cell dies immediately, even if the spell comes from a witch of the own team.

A witch gets "deactivated" when there are one or more witches at Manhattan distance at most two from her, even if there is just one such witch, and from the same team. Deactivated witches do not haunt any sorrounding cells, and farmers and knights can move around them (until the witches get activated again, of course).

Farmers can move to any adjacent empty cell. If that cell is haunted, the farmer dies. Otherwise, the cell is painted with the farmer's team color.

Knights can move to any adjacent or diagonally adjacent empty cell. If that cell is haunted, the knight dies.

A unit killed by the spell of one or more witches will be reborn as a unit of another team. If all the killing witches belong to the same team of the dead unit, the new team will be selected uniformly at random among the rest of teams. Otherwise, the new team will be selected with probability proportional to the number of killing witches of the other teams. For instance, if a unit of player 2 dies under the spell of one witch of team 0, two witches of team 1, and one witch of team 2, then the new team will be 0 with probability $1/3$, and will be 1 with probability $2/3$.

Knights can attack any adjacent or diagonally adjacent rival farmer or rival knight (by an order to move there). Trying to attack an own unit or a witch is an invalid movement. Otherwise, the rival unit loses a random amount of health between 60 and 90. If the new health drops to 0 or below, that unit is captured by the team of the attacking knight.

Every round, more than one order can be given to the same unit, although only the first such order (if any) will be selected. Any player program that tries to give more than 1000 orders during the same round will be aborted.

Every round, all the selected orders will be executed using a random order. For instance, if two farmers try to move to the same empty cell, only the farmer that happens to move first will move there. As another example, assume that one farmer and one rival knight try to move to the same empty cell. If the knight happens to move first, afterwards the farmer will not move, because it would be an illegal movement. However, if the farmer moves first, afterwards the knight will attack the farmer, by trying to move to the farmer's position.

After all the selected movements of a round are played, the units captured by each team are reborn in its corresponding quadrant. Whenever possible, all units are reborn at Manhattan distance at least 3 from witches. Additionally,

farmers are not placed adjacent to rival knights, and knights are not placed adjacent or diagonally adjacent to any rival unit. In the rare cases when this cannot be accomplished, units are reborn on any legal cell in the own quadrant, or on any legal cell in any other quadrant, in even more exceptional situations.

Although there are four players, with numbers 0 to 3, when programming your strategy you must always assume that you are player 0. If you are not, the board will be rotated consistently with your illusion. For instance, if you are player 1, any movement in the `Right` direction by you will be automatically transformed into a movement to the `Top`.

Note that the valid directions are `Bottom`, `BR`, `Right`, `RT`, `Top`, `TL`, `Left`, `LB` and `None`, corresponding to integers from 0 to 8. This circular definition can be used to simplify the implementation of your player. See the `Demo` player for some examples.

If you need (pseudo) random numbers, you must use two methods provided by the game: `random(l, u)`, which returns a random integer in `[l..u]`, and (less frequently) `random_permutation(n)`, which returns a `vector<int>` with a random permutation of `[0..n-1]`.

A game is defined by a board and the following set of parameters:

- *nb_players*: Number of teams in the game (4).
- *nb_rounds*: Number of rounds that will be played (200).
- *nb_farmers*: Initial number of farmers per player (20).
- *nb_knights*: Initial number of knights per player (10).
- *nb_witches*: Constant number of witches per player (2).
- *farmers_health*: The maximum (and initial) health of a farmer (100).
- *knights_health*: The maximum (and initial) health of a knight (200).
- *farmers_regen*: The amount of health a farmer will regenerate when not moving (30).
- *knights_regen*: The amount of health a knight will regenerate when not moving (30).
- *damage_min*: The minimum amount of damage a knight will inflict when attacking (60).
- *damage_max*: The maximum amount of damage a knight will inflict when attacking (90).
- *rows*: Vertical size of the board (37).
- *cols*: Horizontal size of the board (37).

2 Programming the game

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

2.1 Running your first game

Here, we will explain how to run the game under Linux, but it should work under Windows, Mac, FreeBSD, OpenSolaris, ... You only need a recent `g++` version, `make` installed in your system, plus a modern browser like Firefox or Chrome.

1. Open a console and `cd` to the directory where you extracted the source code.
2. Run

   ```
   make all
   ```

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

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

   This starts a match, with random seed 30, of four instances of the player Demo, in the board defined in `default.cnf`. The output of this match is redirected to `default.res`.
4. To watch a game, open the viewer file `viewer.html` with your browser and load the file `default.res`.

Use

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

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

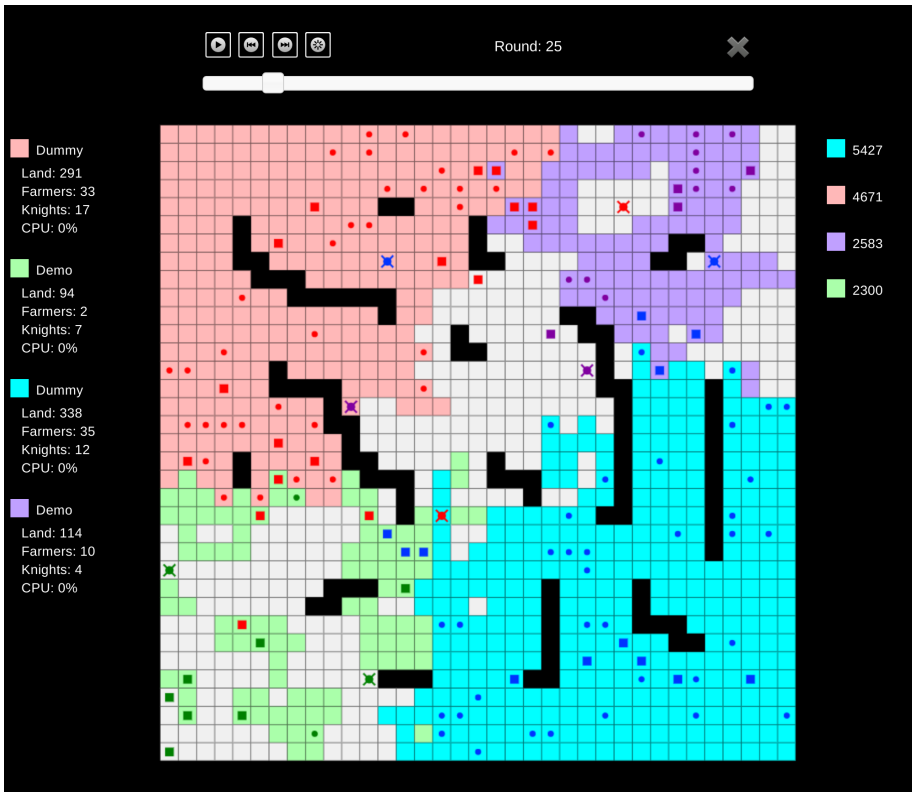
```
./Game --list
```

to show all the recognized player names.

If needed, remember that you can run

```
make clean
```

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



2.2 Adding your player

To create a new player with, say, name `Sauron`, copy `AINull.cc` to a new file `AISauron.cc`. Then, edit the new file and change the

#define PLAYER_NAME Null

line to

#define PLAYER_NAME Sauron

The name that you choose for your player must be unique, non-offensive and at most 12 characters long. This name will be shown in the website and during the matches.

Afterwards, you can start implementing the virtual method *play*(), inherited from the base class *Player*. This method, which will be called every round, must decide the orders to give to their units.

You can define auxiliary type definitions, variables and methods inside your player class, but the entry point of your code will always be the *play*() method.

From your player class you can also call functions to access the state of the game. Those functions are made available to your code using inheritance, but do not tell your Software Engineering teachers because they might not like it. The documentation about the available functions can be found in an additional file.

Note that you must not edit the *factory* () method of your player class, nor the last line that adds your player to the list of available players.

2.3 Restrictions when submitting your player

When you think that your player is strong enough to enter the competition, you can submit it to the Jutge. 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 AISauron.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`.
- 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 can waste part of your limited CPU time.
- Any submission to the Jutge must be an honest attempt to play the game. Any try to cheat in any way will be severely penalized.

3 Tips

- Read the headers of the classes that you are going to use. Do not worry about the private parts or the implementation.
- Start with simple strategies, easy to code and debug, since this is exactly what you will need at the begining.
- Define simple (but useful) auxiliary methods, and *make sure that they work properly*.
- Before competing with your classmates, focus on defeating the "Dummy" player.

- Keep a copy of the old versions of your player. Make it fight against its previous versions to measure the improvements.
- As always, compile and test your code often. It is *much* easier to trace a bug when you have only changed few lines of code.
- Use **cerr** to output debug information, and add `asserts` to make sure that your code is doing what it should do. Remember to remove them before uploading your code, to avoid making it slower.
- When debuging a player, remove the **cerr**s that you may have in others players' code, so as to only see the messages that you want.
- If using **cerr** is not enough to debug some of your code, learn how to use `valgrind`, `gdb` or any other debugging tool.
- Make sure that your program is fast enough. The CPU time that you are allowed to use is rather short.
- Try to figure out the strategies of other players watching several games. This way, you can try to react against them, or even imitate or improve them in your own code.
- Do not give your code to anybody. Not even an old version. Not even to your best friend. We use plagiarism detectors to compare your programs, also against submissions to games of previous years.
- You can, however, share the compiled `.o` files.
- You can submit new versions of your program at any time.
- Do not wait until the last minute to submit your player. When there are lots of submisions at the same time, it takes longer for the server to run the games, and it could be too late!
- And again: Keep your code simple, build often, test often. Or you will regret.