Ork Island

Enric Rodríguez

April 30, 2024

1 Game rules

The power of Sauron, the Lord of the Rings, is spreading throughout Middle Earth. People flee in terror from his bloodthirsty ork soldiers. As a result, once prosperous cities are now abandoned, and vast areas of land have become as lifeless as a desert. Sauron's troops would quickly conquer all Middle Earth... were it not for their own greed and ambition.

In this game, four players compete to conquer an island of Middle Earth for Sauron. The winner of a match is the one that gets the highest score at the end.

The map of the island is represented with a (randomly generated) square board. The cells on this board can be of different types: `WATER`, `GRASS`, `FOREST`, `SAND`, `CITY` or `PATH`. Being an island, the board is surrounded with `WATER`. Moreover, cells of type `CITY` are grouped into rectangles representing cities, hence the name. Similarly, cells of type `PATH` are grouped into sequences that form paths. Paths connect different cities and never cross each other.

In order to conquer the island, each player runs an army of ork soldiers. At each round of the match, players command their orks. Any player that tries to give more than 1000 instructions in the same round will be aborted. An ork can be instructed to remain still or move one cell towards the north, south, east or west direction. If an ork receives more than one instruction, all but the first one are ignored. Orks cannot move to cells with `WATER` (since otherwise they would lose the layer of dirt on their skin). On the other hand, they can move to all other cells. However, when an ork moves, its *health* (an integer value) decreases. Depending on the type of cell where an ork goes, this decrement in health may be different. When an ork reaches a negative health, it dies and regenerates under the control of the same player. Initially all orks have the same health, and when they regenerate they get this same amount of health again.

Each cell of the board can be occupied by a single ork at most. In the particular situation that an ork A attempts to move to a cell where there is already another ork B (who has moved there previously in the same round, or was there

earlier), the following cases are considered:

- If A and B belong to the same player, the instruction is ignored.
- Otherwise there is a fight, after which one of the two orks will die. If the health of A (after the decrement due to the movement) is strictly greater than the health of B, then B dies. Symmetrically, if it is strictly less than the health of B, then A dies. If there is a tie, then the ork that dies is decided randomly with uniform probability, that is, 50%. The ork that dies regenerates under the control of the other player with the initial amount of health. The winner of the fight keeps the same amount of health.

When an ork dies, it regenerates at the next round on a random position on the shore of the island, that is, on a cell adjacent to the sea which is not `WATER`, `CITY` or `PATH`. Similarly, initially all players have their orks randomly distributed on the shore.

At the beginning of a match all cities and paths are empty, i.e., do not have any orks on their cells. However, once the game starts, orks can move to them. Points are then computed as follows. At the end of a round, for each city the number of orks of each player on its cells is counted. If there is a player who has *strictly* more orks on the city than any other player, then this player *conquers* the city; in case of a tie, the conqueror of the city (if any) does not change from the previous round. In any case, for each city *currently* conquered by a player (i.e., currently under their control), this player accumulates a number of points which is *bonus_per_city_cell* () $\times$ the size of the city (that is, the number of its cells); for paths the same applies as for cities, but the number of accumulated points is *bonus_per_path_cell* () $\times$ the size of the path. Finally, for each player, their *graph of conquests* is considered. In this graph, the vertices are the conquered cities, and the edges are the conquered paths that connect conquered cities. For each connected component of the graph with i vertices, additional *factor_connected_component* () $\times 2^i$ points are obtained.

Let us illustrate with an example how the score is computed. Figure 1 shows a screenshot of the game. Blue represents `WATER`, light green represents `GRASS`, deep green represents `FOREST`, light yellow represents `SAND`, deep grey represents `CITY` and light grey represents `PATH`. The orks of a player are identified with small squares of the same color. Conquered cities and paths are filled with a crossed grid of the color of the player that conquered them.

Let us now count the score of the red player accumulated in the current round:

- **Cities:** The red player has conquered cities (from top to bottom) with dimensions 5×2, 6×5, 2×4, 4×2, 3×2, 2×5, 5×5 and 5×2. In total, $(5 \times 2 + 6 \times 5 + 2 \times 4 + 4 \times 2 + 3 \times 2 + 2 \times 5 + 5 \times 5 + 5 \times 2) \times$ *bonus_per_city_cell()* $= 107$ points if *bonus_per_city_cell()* $= 1$.
- **Paths:** The red player has conquered paths (from top to bottom) with sizes 11, 3, 38 and 18. In total, $(11 + 3 + 38 + 18) \times$ *bonus_per_path_cell*$() =$ 70 points if *bonus_per_path_cell*$() = 1$.

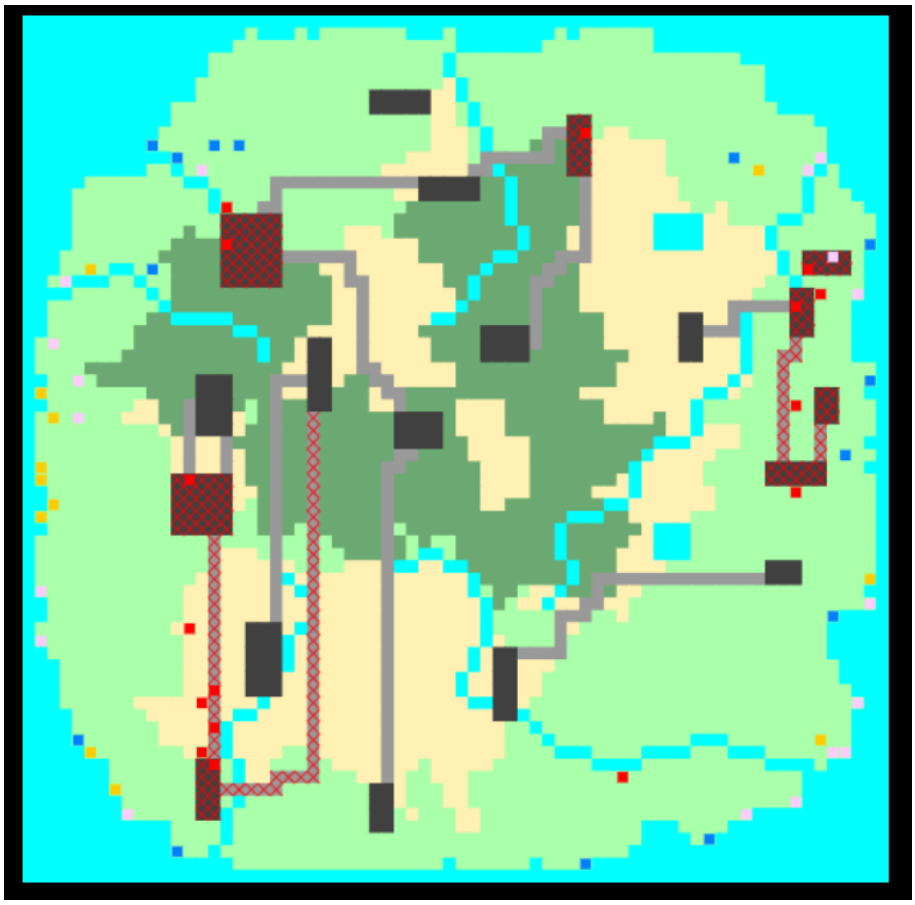

Figure 1: Screenshot of the game.

- **Graph of conquests:** The graph of the red player has five connected components: one with 3 cities, another one with 2 cities, and three components consisting of an isolated vertex. In total, $(2^3 + 2^2 + 3 \times 2^1) \times$ *factor_connected_component*$() = 36$ points if *bonus_per_city_cell*$() = 2$.

Thus, in total the red player has accumulated 213 points in this round.

In general, the execution of a round follows the next steps:

1. All instructions of all players are registered according to the above rules.
2. The instructions are selected randomly and executed (if valid).
3. Dead orks are regenerated.
4. For each player, the points obtained at the end of the round are computed and added to the score.

1.1 Game parameters

A game is defined by a board and the following set of parameters, whose default values are shown in parentheses:

- *nb_players()*: number of players (4)
- *rows()*: number of rows of the board (70)
- *columns()*: number of columns of the board (70)
- *nb_rounds()*: number of rounds of the match (200)
- *initial_health()*: initial health of each ork (100)
- *nb_orks()*: number of orks each player controls initially (15)
- *cost_grass()*: cost in health of moving to a cell of type `GRASS` (1)
- *cost_forest()*: cost in health of moving to a cell of type `FOREST` (2)
- *cost_sand()*: cost in health of moving to a cell of type `SAND` (3)
- *cost_city()*: cost in health of moving to a cell of type `CITY` (0)
- *cost_path()*: cost in health of moving to a cell of type `PATH` (0)
- *bonus_per_city_cell()*: bonus in points for each cell in a conquered city (1)
- *bonus_per_path_cell()*: bonus in points for each cell in a conquered path (1)
- *factor_connected_component()*: factor multiplying the size of connected components (2)

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

2 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.

2.1 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:

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 any file matching `AI*.cc` as a player.
3. 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`.
4. To watch a match, open the viewer file `viewer.html` with your browser and load the file `default.out`. Or alternatively use the script `viewer.sh`, e.g. `viewer.sh default.out`.

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.

2.2 Adding your player

To create a new player with, say, name `Sauron`, copy `AINull.cc` (an empty player that is provided as a template) 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 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.

2.3 Playing against the "Dummy" player

To test your strategy against the "Dummy" player, we provide the `AIDummy.o` object file. This way you still will not have the source code of our "Dummy", but you will be able to add it as a player and compete against it locally.

To add the "Dummy" player to the list of registered players, you will have to edit the `Makefile` file and set the variable `DUMMY_OBJ` to the appropriate value. Remember that object files contain binary instructions targeting a specific machine, so we cannot provide a single, generic file. If you miss an object file for your architecture, contact us and we will try to supply it.

You can also ask your friends for the object files of their players and add them to the `Makefile` by setting the variable `EXTRA_OBJ`.

2.4 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 `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 do not even need to include the corresponding library.
- You cannot open files nor do any other system calls (threads, forks, ...).
- When run in the Jutge.org server, your CPU time and memory usage will be limited. If exceeded (or if your program aborts), your player will not be allowed to execute more instructions.
- 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.

3 Tips

- 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 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, because they make the execution slower.
- When debugging a player, remove the `cerr`s you may have in the other players' code, so as to only see the messages that 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 optimisation.
- 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 analyse the files that the program `Game` produces as output, which describe how the game 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 running locally, use different random seeds with the `-s` option of `Game`.
- Before competing with your classmates, focus on qualifying and defeating the "Dummy" player.
- 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 GIVE YOUR CODE TO ANYBODY.** Not even an old version. We are using plagiarism detectors to compare pairwise all submissions (including programs from previous competitions). However, you can share the compiled `.o` files.
- 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.