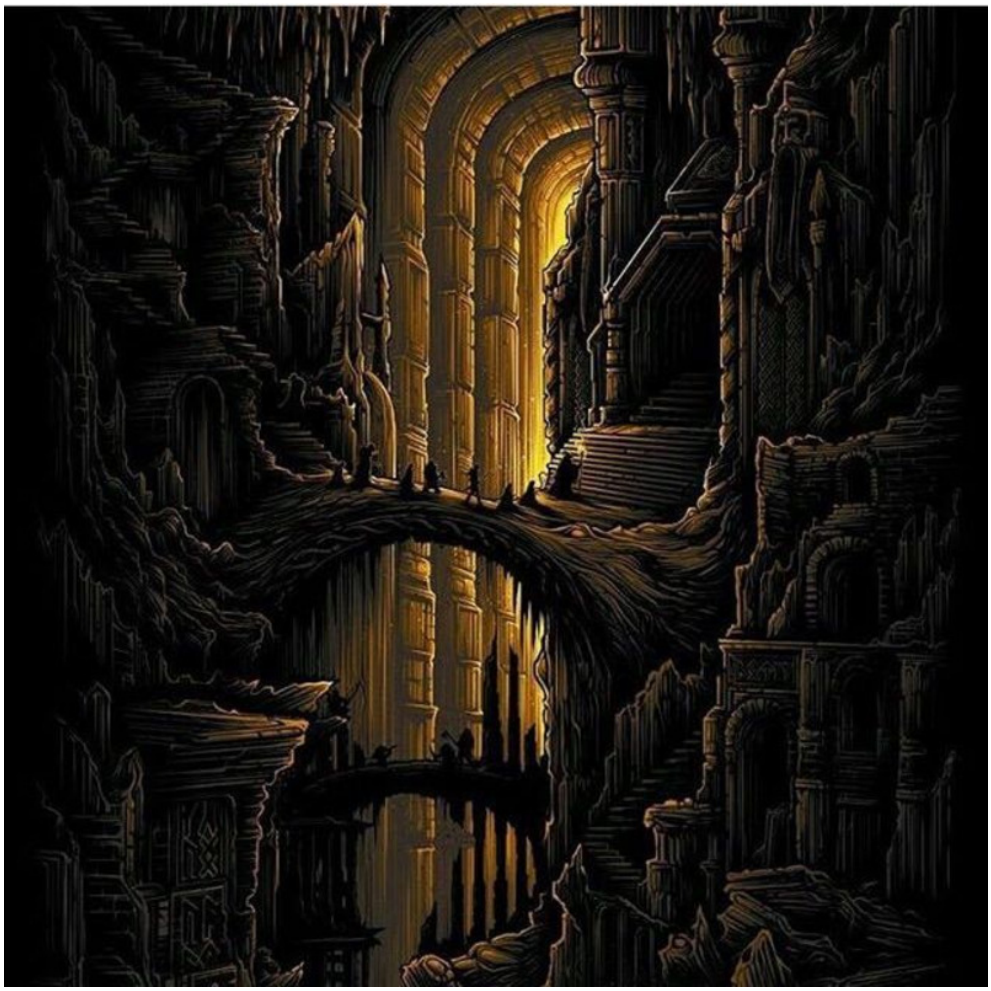
Moria

Salvador Roura

May 2, 2022



1 Game rules

This is a game for four players, identified with numbers from 0 to 3. Each player has control over a clan of dwarves helped by some wizards. The goal of the game is to dominate the ancient kingdom of Moria.

This game also includes some units of Sauron: orcs, trolls, and one Balrog. Sauron units correspond to player -1.

The game lasts 200 rounds, numbered from 1 to 200. Each unit of every player, Sauron units included, can move at most once per round.

The board has dimensions 60×60. Units cannot move outside it. The 2 upper rows, the 2 lower rows, the 2 leftmost columns and the 2 rightmost columns are cells outside Moria, and therefore are the only cells of type Outside.

The rest of cells, of type Cave, Abyss, Granite and Rock, belong to Moria. Cave cells are passable by any unit. Abysses are cells only passable by orcs and the Balrog. Granite and rock cells are only passable by the Balrog. However, rock cells are soft enough to be excavated and converted to cave cells by dwarves.

Initially, most Moria cells are of the rock and cave types. By default, 80 cave cells have a treasure. Dwarves will have to excavate to reach those treasures. There are also some granite cells. Abysses only appear, with small probability, after rock cells are excavated by dwarves.

Each cell can have at most one unit on it. When a unit tries to move onto a cell already occupied by another unit, this in fact means an attack. A unit that attacks does not move during that round, even if the attacked unit gets killed. This includes "attacking" a rock cell (see below).

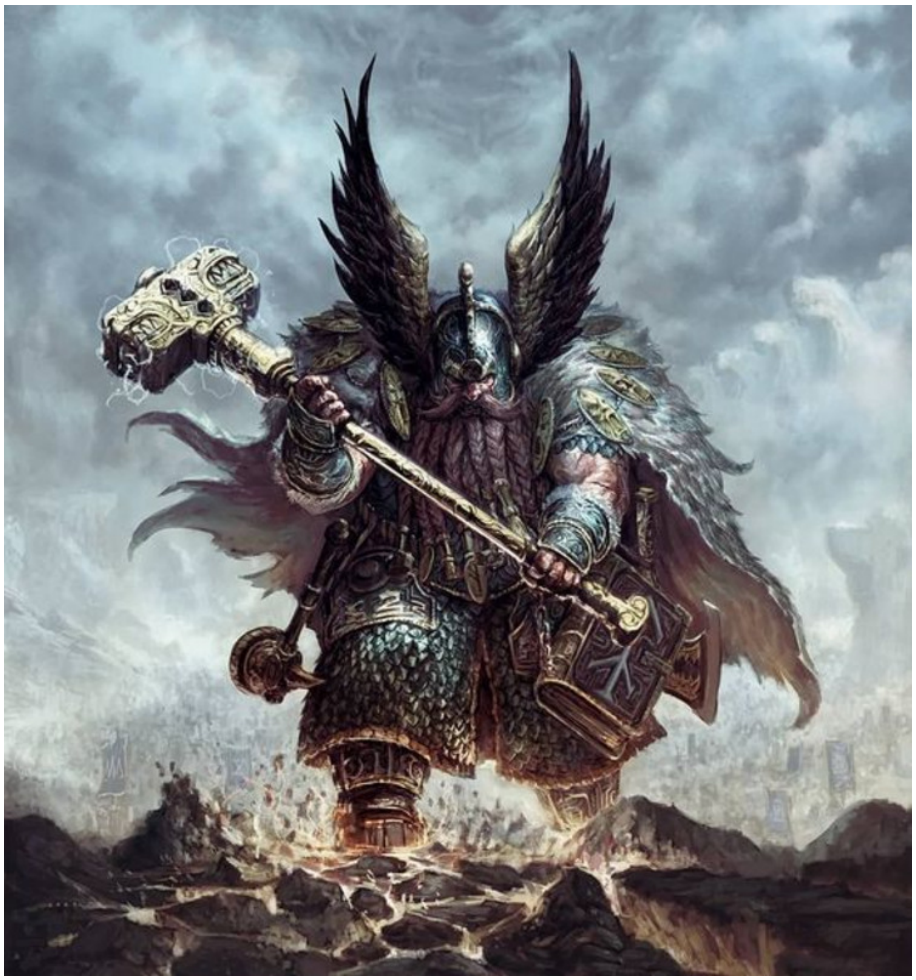
Dwarves and wizards cannot attack units of the same clan. (This would be an illegal move.) Orcs and trolls will never attack other Sauron units. The Balrog does not attack: it just kills everything surrounding it.

Dwarves are the main characters of this game. At the beginning, each clan has 20 of them. They are born with 100 points of health. When they attack, the adjacent enemy unit loses between 20 and 40 points of health. Dwarves can move horizontally, vertically and diagonally on outside and cave cells.

A dwarf can excavate a rock cell by "attacking" it. A rock cell excavated 5 times (by one or more dwarves) becomes an abyss with probability 4%, and a cave cell otherwise.

An abyss is a bottomless hole that is impassable by dwarves, wizards and trolls. Any dwarf or wizard that moves onto an abyss (with no orc in it) falls

into the abyss and dies immediately. Trolls will never try to move onto an abyss. Orcs enter Moria only through abysses, and can afterwards pass over them. The Balrog does not care about the abysses of this game.



When a dwarf moves onto a cave cell (with no unit on it), that cell becomes currently conquered by his clan. If the cave has a treasure, the dwarf picks it. Consequently, the counter of treasures accumulated by the clan of the dwarf increases by one.

At any moment, let c be the number of cells currently conquered by a clan, and let t be the total number of treasures already accumulated by the clan. Then, the current number of points of this clan is $c + 10t$. The clan with the most points after all the rounds have been played wins the game.

At the beginning of the game, each clan has 5 wizards. Like dwarves, wizards can move on outside and cave cells. However, the wizards of this game are weak and slow: they are born with only 50 units of health, they cannot hit any other unit, they cannot conquer any cell, they cannot pick treasures, and they can only move horizontally or vertically, not diagonally. But wizards do have an interesting property: they totally heal any ally unit horizontally or vertically adjacent to them at the end of each round.

Let us now describe the units of Sauron. First, note that the decisions made by orcs, trolls and the Balrog do not depend on the clans of the nearby dwarves and wizards. From the point of view of the Sauron units, all dwarves and wizards equally deserve to die.

Initially, the board has no abysses nor orcs on it. Abysses may be revealed when dwarves excavate rock cells. Afterwards, each abyss "generates" an orc with probability 2% at each round. There is one limitation: the total number of orcs on the board will never exceed 20.

Orcs are born with 75 points of health. An orc attack reduces the health of one adjacent enemy unit between 15 and 30 points. Orcs can move horizontally, vertically and diagonally on caves and abysses. An orc never leaves Moria. After an orc dies, another orc with the same identifier and 75 points of health may enter Moria through an abyss.

Loosely speaking, an orc acts this way: If it has an adjacent dwarf or wizard, the orc attacks him. Otherwise, the orc approaches the nearest dwarf or wizard inside Moria.

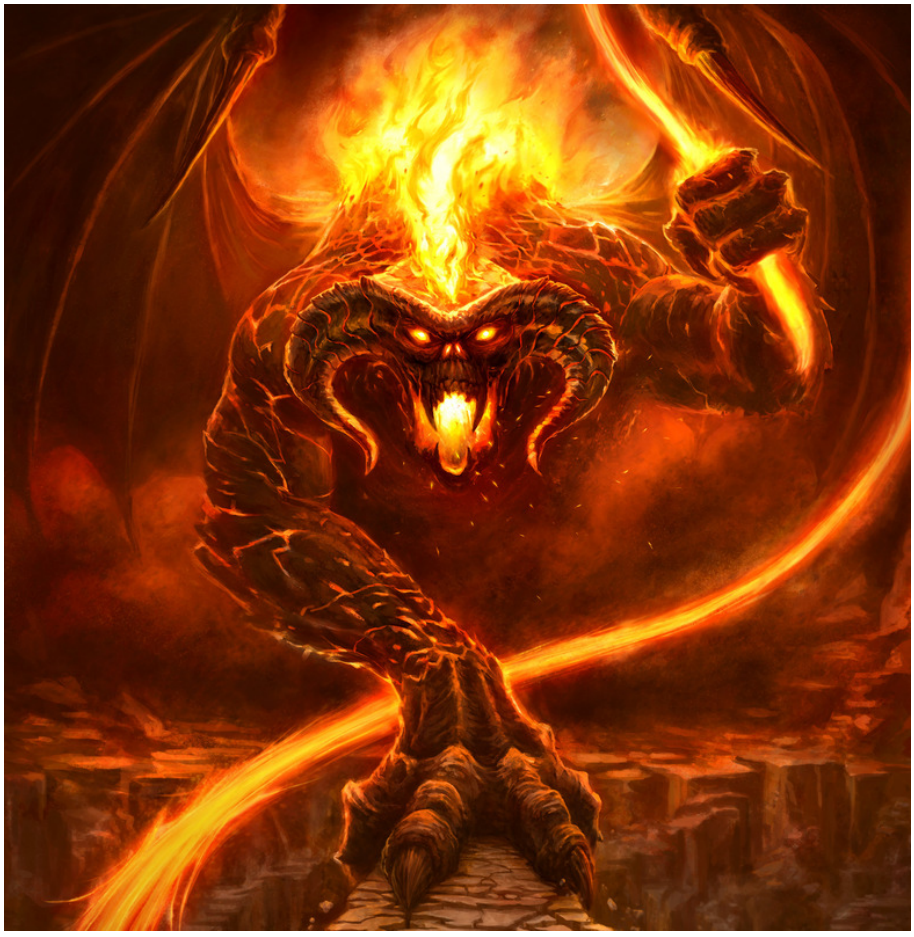
Trolls are not particularly evil. They are just stupid, but very strong. Trolls are born with 500 points of health. When they hit, the adjacent attacked unit loses between 50 and 150 points of health. Trolls move horizontally, vertically and diagonally on outside cells and caves. Trolls will never leave Moria on purpose (once in Moria, they will stay), but they are reborn outside it. The game always has 4 trolls.

Roughly speaking, a troll behaves like this: If the troll has an adjacent dwarf or wizard, it attacks him. Otherwise, if the troll is outside Moria, it tries to get inside by approaching the nearest cave cell. Otherwise, the troll moves to a randomly chosen adjacent cave cell.

In addition to their general behavior, orcs and trolls always try to avoid the Balrog, but they are not very smart at that.

Gandalf is not here, so the Balrog is immortal. It always approaches the nearest dwarf or wizard inside Moria. Any time, all units horizontally, vertically and diagonally adjacent to the Balrog (this includes orcs and trolls) immediately

get killed. The Balrog cannot move diagonally nor leave Moria, but it can visit every cell inside.



Sauron units do not pick any treasures. However, orcs and trolls "unconquer" all visited cave cells. The Balrog "unconquers" all surrounding cave cells (up to eight).

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, the selected movements of the four players will be executed using a random order, but respecting the relative order of the units of the same clan. For instance, if several dwarves walk in a single file, and there are not other units around interfering, they all can move one step forward with no collisions among them, by ordering movements from the front of the file to its back.

As a consequence of the previous rule, consider giving the orders to your units at every round from most urgent to less urgent.

Note that each movement is applied on the board resulting of the previous movements. As another example, suppose that one wizard and three dwarves (let us call them X, Y and Z) of the same clan try to move in this order on an abyss occupied by an orc. First, the wizard will not move because the target cell is occupied. Then, X will attack the orc. Assume that the orc gets hurt but not killed. Then, Y will also attack the orc. Suppose that the orc gets killed now. Finally, Z will try to move on an abyss, and he will tragicomically die.

After all the selected movements of the fours players have been executed, it is the turn of Sauron: orcs, trolls and the Balrog will move, in this order.

If a Sauron unit has several movements that are equally attractive to it, it will choose one at random. For instance, if an orc or a troll has several adjacent dwarves and wizards, it will just choose one of them at random and attack him. Similarly, if the Balrog has several dwarves and wizards at the same distance inside Moria, it will choose one of them at random and approach him.

After all the selected movements of a round are played, the killed dwarves, wizards and trolls are reborn. By default, this is done on outside cells, with their respective maximum health points. In rare cases where no safe enough cells can be found, they can be reborn on any treasureless cave cell. If possible, all these units are initially generated on treasureless cave cells.

A dwarf or wizard killed by another clan will be "captured", so the reborn unit will belong to the attacking clan. A dwarf or wizard that has been killed by a Sauron unit or that has fallen into an abyss will be assigned to a randomly chosen different clan.

At the very end of each round, the units horizontally or vertically adjacent to ally wizards fully recharge their health points.

As a result of all the above rules, the total number of dwarves, wizards and trolls remain constant during the whole game: 80, 20 and 4, respectively.

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]`.

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.

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. If, for example, you are using a 64-bit Linux version, run:

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

   ```
   cp Board.o.Linux64 Board.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 as a player any file matching `AI*.cc`.
4. 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`.
5. 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

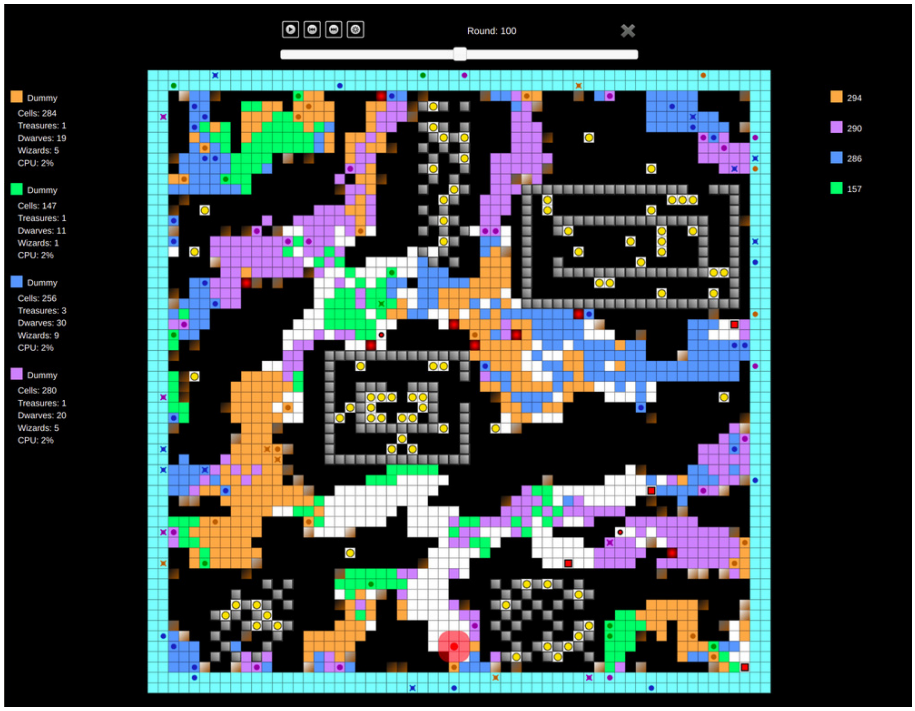
```
./Game --list
```

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 `Gimli`, copy `AINull.cc` (an empty player that is provided as a template) to a new file `AIGimli.cc`. Then, edit the new file and change the

#define PLAYER_NAME Null

line to

#define PLAYER_NAME Gimli

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 your 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 the additional file `api.pdf`.

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 AIGimli.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 that 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.
- 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.

3 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.