Pandemic

Martí Oller

January 27, 2026

1 Game Rules

The game is set in the dystopian year 2025, when a new virus emerged. The new virus was the last straw for an already frail civilization. As a consequence, society collapsed, the stock market crashed and, most tragically, billions of people died. Out of the few survivors, four rival factions have emerged and are fighting against each other for the control of a society left in shambles. Who will win?

In this game, four players compete to dominate the ruins of society. The winner of the match is the one that gets the highest score in the end.

The game is played on a map represented by a (randomly generated) square board. The cells of the map can be of four different types: `WALL`, `GRASS`, `CITY` or `PATH`. Cells of the type `CITY` are grouped into rectangles representing cities, hence the name. Similarly, `PATH` cells are grouped into sequences that form non-intersecting paths that connect cities. Finally, there are `WALL` cells surrounding the map, and some additional interior ones resembling fallen cities and paths that are now inaccessible.

Each player controls a number of units, which can be moved every round. Any player that tries to give more than 1000 instructions in the same round will be aborted. Units can remain still or move in any of the four cardinal directions (top, right, bottom or left), as long as there isn't a wall. If a unit receives more than one instruction, all but the first will be ignored.

Cells can be occupied by at most one unit. Hence, a unit A can't just simply move to a cell where there is another unit B. If A tries to move to a cell occupied by B, and both units belong to the same player, nothing will happen (A will not move). However, if they belong to different players, then A will attack B, and B will lose between 25 and 40 points of health (randomly with uniform probability). If B reaches a negative amount of health due to the attack, unit B dies and regenerates under the control of unit A's player, and A moves to B's position. Otherwise, B survives and A doesn't move after the attack.

Every five rounds, a mask spawns in a grass cell. Masks are represented in the map as small black dots. If a unit that doesn't have a mask moves to a cell where there is a mask, it will pick it up and wear it automatically. If the unit is already wearing a mask, it will just ignore it. If a unit wearing a mask dies, the mask disappears.

Points are obtained by conquering cities and paths. Initially, they are all empty and have no owner. As the game progresses, units can move there and conquer them. At the end of every round, the number of units of each player in every city and path is computed. If a player has strictly more units than any other player in a given city or path, the player conquers it. In case of a tie, the owner of the city or path doesn't change from the previous round.

Every round, points are added to each player depending on the cities and paths they own. For every city under the control of a player, a total of bonus_per_city_cell()$\times$ size of the city points are added to her. Similarly, every path gives the player a total of bonus_per_path_cell()$\times$ size of the path. Additionally, for each player, her graph of conquests is considered. In this graph, the vertices are the cities of that player, and the edges are conquered paths connecting conquered cities. For each connected component of that graph with size i, additional $2^i\times$ factor_connected_component() points are added.

For instance, consider the state given by the screenshot in Figure 1. The red player will obtain the following number of points:

- The total size of controlled cities is $16+20+24+10+12+25+30 = 137$. Hence, the red player will obtain 137 points if bonus_per_city_cell() $= 1$.
- The total size of controlled paths is $3+7+6+4+9+12+21 = 62$. Hence, the red player will obtain 62 points if bonus_per_path_cell() $= 1$.
- Finally, the graph of conquests has a component of size 3, a component of size 2 and two components of size 1, accounting for $2\times(2^3+2^2+2\times 2^1) =$ 32 points if factor_connected_component() $= 2$.

In total, the red player will receive 231 points this round.

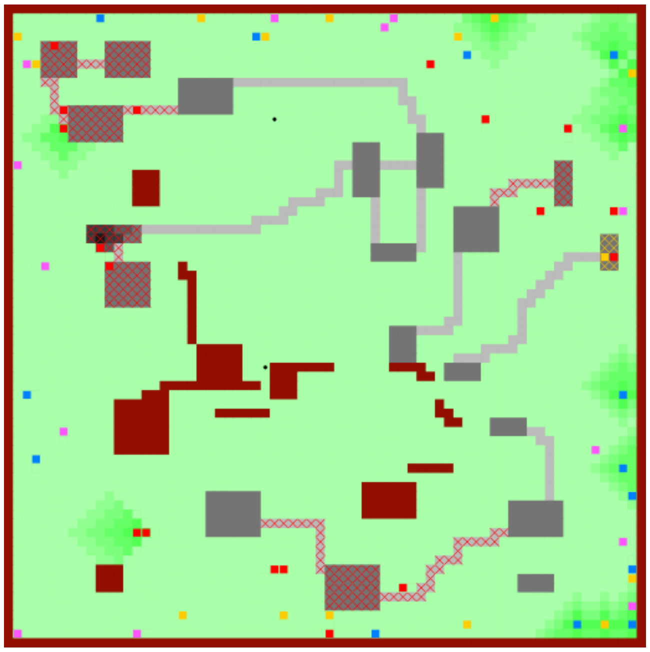

Figure 1: Screenshot of the game

Some of the units are infected with a potentially deadly virus, which they can spread to locations near them.

A unit that is infected by the virus will lose a fixed amount of health every round, which can vary between 2 and 5 (but is fixed for a given infected unit). If the unit's health points decrease to a negative number due to the virus, the unit dies and respawns under the control of a random player (decided randomly with uniform probability 1/4). However, units may also recover from the virus. If a unit has been infected by the virus for t turns, the probability p of healing from the virus is computed as follows:

$$p = \min\left(1,\ 0.001\left(\frac{t^2}{16} + 1\right)\right)$$

What this formula means is that a unit that has had the virus for longer is more likely to recover from the infection. In practice, the formula implies that roughly 50% of infections will disappear within the first 31 turns, and approximately 90% of them will have disappeared by the 47th turn (provided the unit is still alive).

A unit that recovers from the virus becomes immune from the infection, and cannot catch the virus again.

Infected units can also propagate the virus to their surroundings. Every cell c has a certain amount of virus in it, which is represented by an integer `c.virus` initially set to zero. In the map, cells with higher concentrations of the virus are represented by darker colors. Every turn, `c.virus` is updated using the following procedure:

1. For all cells c that contain non-masked infected individuals, their amount of virus, as stored in `c.virus`, is increased by 3. This doesn't happen if the infected unit is wearing a mask.
2. For all non-wall cells in the board, their virus amount is updated by taking the maximum of the following five (at most) numbers:
   - The current virus amount in the cell minus 1.
   - The current virus amount in any of the neighboring cells minus 1, but not all neighbors are considered: for grass cells, only neighbors that are also grass are considered, for cities only neighboring cells that are cities or paths are considered, and for paths also only cells that are cities or paths. In other words, the virus doesn't travel through walls and doesn't propagate from grass to cities or paths and viceversa.
3. The virus amount is capped at 4 in grass cells, and at 10 in cities and paths, and must be non-negative in all cases.

Units can get infected by being in cells with non-zero amounts of virus. The probability p_1 that a non-masked, non-immune unit will catch the virus in a cell with virus amount v is $p_1 = v/\text{factor\_infection}()$. If the susceptible unit is wearing a mask, that probability becomes $p_2 = p_1/\text{mask\_protection}()$. With the default parameters, these numbers are $p_1 = v/50$ and $p_2 = v/1000$.

Immune units can't be reinfected, and units that are currently infected can't get reinfected in any way.

Every time a unit gets infected by standing in a cell with virus in it, the amount of health they will lose in each turn is set to a number between 2 and 5 with uniform probability. This amount is called the damage, and does not change for a given infected unit.

When a unit dies (either due to an attack or the virus), it will spawn with full health in a border of the map. All spawned units have a 20% chance of having the virus, and the damage is set to a number between 2 and 4 (not 5, in this case) with uniform probability.

Initially, at round 0, exactly three units of each player will have the virus, with virus damage set to 2, 3 and 4, respectively.

In general, the execution of a round is done following these steps:

1. All instructions of all players are registered according to the above rules.
2. The instructions are randomly sorted and executed (if valid).
3. The virus is propagated by the infected units to their surroundings following the aforementioned rules.
4. Units may get infected or healed from the virus, and infected units take damage from the virus.
5. Dead units are regenerated.
6. If the round number is a multiple of 5, a mask spawns in a grass cell.
7. For each player, the points obtained at the end of the round are computed and added to the score.

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

- nb_players(): number of players (4).
- rows(): number of rows (70).
- cols(): number of columns (70).
- nb_rounds(): number of rounds (200).
- initial_health(): initial health of each unit (100).
- nb_units(): number of units each player controls initially (15).
- 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).
- infection_factor_(): factor dividing the probability of infection (50).
- mask_protection_(): factor dividing the total probability of infection when wearing a mask (20).

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

2 Programming the game

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

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

   to copy the object file for player "Dummy" (copy `AIDummy.o.MacOS` if you use Mac).
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` 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 you can use. Particularly useful is

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

to show all the recognized player names. If needed, remember 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 `Manolito`, copy `AINull.cc` (an empty player that is provided as a template) to a new file `AIManolito.cc`. Then, edit the new file and change the

#define PLAYER_NAME Null

line to

#define PLAYER_NAME Manolito

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

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
```

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

2.4 Restrictions when submitting a 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 `AIManolito.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.

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.