Àlex Moré Guardiola

January 27, 2026

1   Rules

Poquémon is a game based on the Japanese video game saga Pokémon[1].

Each player controls a number of Poquémon. The goal of a player is to get the maximum score by collecting point bonuses and killing opponent Poquémon.

A match of the game consists of a number nb_rounds() of rounds. In each of these rounds, Poquémon can move around a rectangular board. Cells in this board may be occupied by a Poquémon or contain bonuses of several kinds, or be a wall (which Poquémon cannot cross), or be empty.

Whenever a Poquémon moves to a cell with a bonus (for instance, a point bonus), it collects it; in this case, after some rounds the bonus appears again in a random cell of the board.

The board will always be surrounded by walls. Some walls may appear and disappear along the game. These are called ghost walls and cannot be placed in the perimeter. These ghost walls are located on the board and may be present or hidden. Every wall_change_time() rounds every ghost wall will change its state (present, or not present) but it is not necessary that all board's ghost walls change at the same round. In other words, they have the same period but

---

[1]More info in http://www.pokemon.com/us/

not the same phase. If a Poquémon is located in a position where a ghost wall appears, it dies. To know if there is a ghost wall at a certain position P you can use ghostWall(Pos p) function, which returns the remaining rounds to change the state of the wall or -1 if this cell is not a ghost wall.

In each round, a Poquémon can also attack another Poquémon and wage a battle. As a result, the attacked Poquémon can die. Each Poquémon has some attributes that are considered in a battle: attack, defense and scope. These attributes can be improved by collecting respective bonuses (attack bonus, defense bonus, scope bonus and stone bonus, a special bonus that improves all Poquémon's stats). The scope of a Poquémon and the number of stones can be collected are limited.

Any dead Poquémon will appear again after player_regen_time() rounds in a position that guarantees that an action can be performed in the next round.

As pointed out above, there are two ways to get points. A player can collect point bonuses of the board or kill opponent Poquémon and win the battle_reward() percent of the total points of the opponent.

- The following list explains the parameters that configure a game:

  - nb_players(): Number of players.
  - nb_pokemon(): Number of Poquémon per player.
  - nb_rounds(): Number of rounds in the game.
  - nb_ghost_wall(): Number of ghost walls on the board.
  - nb_point(): Number of point bonuses on the board.
  - nb_stone(): Number of stone bonuses on the board.
  - nb_scope(): Number of scope bonuses on the board.
  - nb_attack(): Number of attack bonuses on the board.
  - nb_defense(): Number of defense bonuses on the board.
  - player_regen_time(): Time (in rounds) before a Poquémon appears again after dying.
  - wall_change_time(): Time (in rounds) before a ghost wall changes its status, present or hidden.
  - point_regen_time(): Time (in rounds) before a point bonus appears again after having been taken.

- stone_regen_time(): Time (in rounds) before a stone bonus appears again after having been taken.
- scope_regen_time(): Time (in rounds) before a scope bonus appears again after having been taken.
- attack_regen_time(): Time (in rounds) before an attack bonus appears again after having been taken.
- defense_regen_time(): Time (in rounds) before a defense bonus appears again after having been taken.
- battle_reward(): Percent of points the attacker will get of the total points of the defender if it wins the battle.
- max_scope(): Maximum scope that a Poquémon can reach.
- max_stone(): Maximum number of stones bonuses that a Poquémon can take.
- rows(): Number of rows of the board.
- cols(): Number of columns of the board.

- The different kinds of cells of the board are:

  - Empty: Empty cell.
  - Wall: Cell with a wall (ghost wall or not).
  - Point: Cell with a point bonus.
  - Stone: Cell with a stone bonus.
  - Scope: Cell with a scope bonus.
  - Attack: Cell with an attack bonus.
  - Defense: Cell with a defense bonus.

- Each cell can be visited by at most one Poquémon. Each Poquémon can be alive or (temporarily) dead.

- The first round is the round 0.

- Initially all Poquémon will have one point of attack, one point of defense and one point of scope. These attributes can be upgraded by collecting their respective bonuses and will be used to win battles against opponent Poquémon.

- Each round, each player can ask only for one action for each of their Poquémon. This player can choose –independently of the other players—

what their Poquémon have to do: moving to an adjacent position, throwing an attack OR nothing. An attack will only be accepted if when it is thrown, there is an opponent Poquémon to receive it. Otherwise, the action will be considered as null. If a player asked for more than one action with one of their Poquémon, only the first one will be accepted.

- The available directions to move and attack are top, bottom, left and right. A Poquémon cannot move to a cell with a wall.

- If a Poquémon tries to go to a cell occupied by another Poquémon, the movement will not be performed.

- In Poquémon's game there are the following kinds of bonuses:

  - Point: Increases player's score.
  - Attack: Increases Poquémon's attack attribute.
  - Defense: Increases Poquémon's defense attribute.
  - Scope: Increases Poquémon's scope attribute.
  - Stone: Increases some Poquémon's attributes.

- To take a bonus from the board, it is only necessary to move a Poquémon to this cell of the board.

- Which are the consequences of collecting each bonus?

  - If a Point bonus is collected, the player adds to their scoreboard the value of this bonus. The value of the point bonus can be asked by using pointsValue(Pos p) function, which returns the number of points of this cell (100, 200, 300, 400 or 500) or -1 if there are not any point bonuses.
  - If an Attack or Defense bonus is collected, the Poquémon will receive one point of the corresponding attribute.
  - If a Scope bonus is collected, the Poquémon will receive one point of scope except if its scope is == to max_scope(). In this case, this bonus will not have any effect. Scope lets a Poquémon attack farther. The value of this attribute is the number of cells away a Poquémon can attack.

- If a Stone bonus is collected, the Poquémon will receive two points of attack, two points of defense and one point of scope (the latter only if the scope of the poquémon is $<$ than max_scope()). The maximum number of Stones a Poquémon can collect is max_stone(). When a Poquémon collects more Stone than max_stone() this Poquémon will not increase any attribute.

- Each bonus will appear again on the board in a random cell after point_regen_time(), stone_regen_time(), scope_regen_time(), attack_regen_time() or defense_regen_time() rounds, respectively.

- A Poquémon can only see opponent Poquémon when they are at the same row or column and there are not any walls between them.

- When two Poquémon are aligned at the same row or column, and there is not any wall between them, a battle can take place if one of them asks for it. If the distance between the attacker and the defender is larger than the scope of the attacker, the battle does not take place.

If there are more than two Poquémon aligned and the scope of the attacker is enough to hit more than one opponent, only the closest one will receive the attack.

The result of a battle will be computed following the next rule:

Let a be the attack attribute of the attacker, and d the defense attribute of the defender.

If $a \geq d$ then the attacker's attack updates to max(1, attack-1) and wins battle_reward() percent points of the total points of the defender (rounding down). The defender dies keeping the same attributes (attack, defense, scope) and the same score.

On the other hand, if $a < b$, the attacker's attack updates to max(1, attack-1), the defender's defense updates to max(1, defense-1) and the game

continues.

- Only one attack can be executed for each round. In the situation that more than one Poquémon asked to attack, the final attacker will be decided randomly.

- The actions requested by the players will be executed in the following way: Firstly we will determine a random order of execution among all players. Then, following this order, the actions will be executed. If a Poquémon attacks another Poquémon that has moved before and the scope of the attacker is not large enough the attack will fail and the attacker will not do any action.

- At the end of each round the score of each player will be updated and bonuses and Poquémon will be regenerated if appropriate. Regenerated bonuses will appear in a random position and regenerated Poquémon in a random safe position (a position where the regenerated Poquémon cannot find opponent Poquémon for at least one round). Finally, the Ghost walls with attribute time == 0 will change their state.

- When the game is over, the player with the highest score will be the winner.

2 Viewer

In the following image we can see a screenshot with most of the elements that are present in the game:

- On the top of the window, we can see some buttons that will allow us to pause/play, go to the beginning and go to the end of the game, deactivate the animation mode or close the viewer. A horizontal slide indicates the round number the game is. A help window will be opened by clicking 'h'. This help explains the keyboard shortcuts that control the viewer.

- The scoreboard is on the left and the right of the board. Each player has his name and avatar. The scoreboard indicates his score, the consumed CPU and his attribute status: Attack, Defense, and Scope (when the player becomes froze, a red 'OUT' appears).

- A fine circle surrounding a Poquémon means that the Poquémon is resurrecting.

- In the screenshot, the red Poquémon is attacking.

- Other elements that appear in the screenshot:

- Attack
- Defense
- Scope
- Stone
- Points (with the printed value)

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 HTML5/Javascript viewer to watch them in a nice animated format. Also, a "Demo" player is 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 Chromium.

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 the Makefile will identify as a player any file matching the expression "`AI*.cc`".

3. The call to `make` should create an executable file called `Game`. This executable allows you to run a match as follows:

   `./Game Demo Demo Demo Demo < default.cnf > default.res`

   Here, we are starting a match with 4 instances of the player "Demo" (included with the source code), with the game configuration defined in "`default.cnf`". The output of this match will be stored in "`default.res`".

4. To watch the match, open the viewer (`viewer.html`) with your browser and load the "`default.res`" file.

A script `run.sh` for carrying out steps 2-4 automatically is also provided.

Use the `--help` option of `Game` to see a list of all options you can use. For instance, the option `--list` will show a list with all the available player names.

If needed, remember you can run `make clean` to delete the executable and all object files and start over the build.

3.2 Adding your player

To create a player, copy the file `AINull.cc` (an empty player that is provided as a template) to a new file with the same name format (`AIWhatever.cc`).

Then, edit the file you just created and change the playername line to your own player name, as follows:

#define PLAYER_NAME Whatever

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 Board class in `Board.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 via the class Player in `Player.hh` . The documentation on the available functions can be found in the aforementioned header files of each class. 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 file `Utils.hh` for useful data structures.

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 available players.

3.3 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.

Pro tip: You can ask your friends for the object files of their players and add them to the `Makefile` too!

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 (`AIWhatever.cc`).

- Your code cannot use global variables (use attributes in your class instead).

- You are only allowed to use standard libraries like `vector`, `map`, `cmath`...

- Your code cannot open files nor do any other system calls (threads, forks...).

- Your CPU time and memory usage will be limited when executed on Jutge.org. The time limit is 1 second for the execution of the entire game. If the time limit has been exceeded (or if the execution of your code aborts), your player will be frozen and will not admit further instructions any more.

- Your program should not write to cout nor read from cin. You can write debug information to cerr (but remember that doing so on the code you upload can waste part of your limited CPU time).

4 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 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, because they make the execution slower.

- 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 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 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.

- 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!

- Most of the game parameters (number of rounds, ...) will not change, but if your strategy can adjust to them, you will be extra-safe in case some changes are needed.

- You can submit new versions of your program at any time.

- If you create your own board for the game, send it to us before the competition starts and maybe we will include it!

- And again: Keep your code simple, build often, test often. Or you will regret.