Apocalypse Now 2018

Omer Giménez, Jordi Petit, Enric Rodríguez, Salvador Roura

April 30, 2024

1 Game rules

- Four players, each of them controlled by a program, play on a square board with 60×60 cells. The goal in the game is to conquer the maximum number of control points (henceforth called *posts* for the sake of brevity). There are 16 such posts distributed on the board.
- Each cell of the board is identified with a pair (i, j), where $0 \leq i, j < 60$. The i axis goes from north to south, and the j axis from west to east.
- Each player starts in a quadrant of the board: players 0, 1, 2 and 3 are initially in the north-west, north-east, south-east and south-west quadrants, respectively.
- There are three kinds of units: *soldiers*, *helicopters* and *parachuters*. (In fact, a parachuter is only the possibility of throwing a new soldier, who still does not exist, from a helicopter. So strictly speaking parachuters as such never exist.) When a match starts, each player has 20 soldiers, 2 helicopters and 0 parachuters. Soldiers start in their quadrant, randomly distributed. Helicopters of player 0 start oriented eastwards and southwards, respectively. Helicopters of the other players are oriented symmetrically.
- There are four kinds of land: *forest*, *grass*, *water* and *mountain*. All the perimeter of the board is surrounded by mountains. Soldiers can only be in forests or grass. Helicopters can be in any kind of land except for mountains (we can imagine they are too high). Initially, the land on which soldiers and helicopters are is consistent with these rules. Soldiers and helicopters move in different planes (heights), and therefore never interfere.
- A match lasts 200 rounds. At each round, the four players give instructions to their units. If a soldier or helicopter gets more than one instruction in a round, only the first one is considered for execution; the rest are ignored.

- At the end of each round all instructions of all players are considered for execution, in the following order: first those of parachuters, then those of helicopters, and finally those of soldiers. Inside each of these groups, instructions are executed in a random order, without giving priority to any of the players.
- Soldiers do not move in a round if they do not get any instruction, if the place they should move to is not allowed (there is water or mountain), or if in the moment of trying to execute the instruction, another soldier (of the same player or an enemy) occupies the place they should move to.
- Similarly, a helicopter does not move in a round if it does not get any instruction, if the place it should move to is not allowed (there is mountain), or if in the moment of trying to execute the instruction, another helicopter (of the same player or an enemy) occupies the place they should move to.
- Soldiers occupy one cell. Soldiers can try to move to any of the adjacent cells, be it horizontally, vertically, or in diagonal. If the cell they try to move to is occupied by a soldier of another player, the enemy gets a random damage depending on the land he is: between 20 and 40 points if he is in a forest (and so it is relatively protected), and between 50 and 100 points if he is on grass (and so it is relatively unprotected). Each soldier initially has 100 life points. When all are lost, the soldier dies. Soldiers of the same player never attack each other.
- Helicopters occupy a square of 5×5 cells of sky. At each round, each helicopter can try to move one or two cells forward (in the direction and sense in which it is oriented). If any of the 5 (or 10) new cells it should move to is occupied by another helicopter, or if any is mountain, the helicopter does not move (or moves only one of the two steps, if the first step is possible but the second is not). Helicopters never attack each other, and cannot get any kind of damage. So each player has 2 helicopters all along the match.
- Helicopters can also turn around 90 degrees clockwise, turn around 90 degrees counter clockwise, or throw napalm. Turns are always valid. On the other hand, napalm can only be thrown if it is available: each helicopter has to wait a minimum of 30 rounds from the last throw of napalm (or from the beginning of the match).
- When a helicopter throws napalm, the square 5×5 of land which is below it is set on fire, and all soldiers on it, no matter of which player, die. Moreover, fire burns for 5 rounds on grass and water, and 10 rounds on forest. After burning, forests become grass.
- At the end of each round, fire propagates according to the following rule: any cell with forest and without fire that is horizontally or vertically adjacent to one or more cells with fire, has a probability of 10 per cent to start burning. If that is the case, fire lasts 10 rounds (at least: when napalm is

thrown on a cell that is already burning, the counter of duration of fire is reset to the maximum).

- A soldier dies when he moves to a cell of grass or forest with fire, or when he is on a cell of forest on which fire is propagated.
- Each soldier or helicopter has a positive integer that uniquely identifies it. When a soldier dies, he disappears (included the identifier), and a new parachuter of another player appears (according to what is explained below), who can later become a soldier. Therefore, the sum of soldiers and parachuters along the match is always 80.
- Each helicopter has a certain number of parachuters available. Each of these allows making a healthy soldier with a new identifier appear below the respective helicopter. Even though a player can have many parachuters available, at each round at most 4 parachuters can be thrown. Moreover, parachuters may expire: an unused parachuter in 20 rounds dies. Additionally, a parachuter who is thrown on a cell which is not covered by the 5×5 square of the corresponding helicopter, or on a cell with water, or with fire, or with a soldier (own or enemy) also dies.
- When an enemy soldier is killed, be it by attacking him with an own soldier or by throwing napalm, the attacking player gets a new parachuter.
- Any soldier that dies due to moving into a fire, or by the propagation of a fire, or by napalm thrown by the own player, becomes a parachuter of one of the other three players, chosen uniformly at random.
- Expired parachuters and those that die when they are thrown also become parachuters of another player. The new player is chosen always at random, with an exception: if a parachuter is thrown over a soldier of another enemy player, the new parachuter is added to this player.
- When a new parachuter is added to a player, the helicopter it becomes assigned to is determined randomly.
- When a soldier moves to a post, or when a parachuter falls on it (if the post does not have any soldier), the post is conquered. Posts can be conquered and reconquered by several players as many times as wished. Initially, posts do not belong to any player.
- According to their strategic location, posts may have a high value (100 points) or a low value (50 points). At each quadrant there are 2 posts of 100 points, and 2 other posts of 50 points. The points achieved by a player in a round are computed as follows: each post that the player controls at the end of the round gives as many points to that player as its value, and each soldier (no parachuter) of the player gives 1 additional point. These points are accumulated to the score of the player in the previous round.

1.1 Game parameters

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

Parameter	Value	Description
NUM_PLAYERS	4	Number of players
GAME_ROUNDS	200	Rounds that the game lasts
MAX	60	Number of rows and columns of the board
NUM_SOLDIERS	20	Initial number of soldiers per player
NUM_HELICOPTERS	2	Number of helicopters per player
NUM_POSTS	16	Total number of posts
LIFE	100	Initial life points of soldiers
FOREST_DAMAGE	20	Minimum damage per attack at forest (maximum damage: the double)
GRASS_DAMAGE	50	Minimum damage per attack at grass (maximum damage: the double)
ROUNDS_NAPALM	30	Rounds before napalm is available again
REACH	2	Napalm is thrown in the square (2**REACH*+1)$\times$ (2**REACH*+1) centered at the helicopter
FOREST_BURNS	10	Rounds that a forest cell burns
OTHER_BURNS	5	Rounds that any other cell burns
PROB_FIRE	10	% of probability of propagation of fire
ROUNDS_JUMP	20	Maximum number of rounds to jump
MAX_JUMP	4	Maximum number of parachuters of a player who can jump in the same round
HIGH_VALUE	100	Value of the most valuable posts
LOW_VALUE	50	Value of the least valuable posts

Unless there is a force majeure event, the above values of the parameters are the ones that will be used in all matches of 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 `MyPlayer`, copy `AINull.cc` (an empty player that is provided as a template) to a new file `AIMyPlayer.cc`. Then, edit the new file and change the

#define PLAYER_NAME Null

line to

#define PLAYER_NAME MyPlayer

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 *Player* class in `Player.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.

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 `AIMyPlayer.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, ...).
- When run in the Jutge.org server, your CPU time and memory usage will be limited.
- 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.