Mad Max

Ivan Geffner Salvador Roura

December 13, 2018

1 Game rules

In this game, 4 players have control over an army of warriors and cars on a 60×60 board with 8 cities. The goal of the game is to "conquer" as many cities as possible.

Teams are identified with numbers from 0 to 3. Initially, every team owns 2 of the cities. Every round, the number of cities of each team is added to the corresponding score. The winner of the game is the player with the highest score after the last round.

At the beginning of the game, each team has 23 units: 20 warriors and 3 cars. The warriors start inside the cities of their teams, with at least one warrior in each city. The cars start at the edges of the board, each on a road.

The board has six kinds of cells: Desert, City, Road, Water, (fuel) Station, and Wall. Warriors can move on desert, cities and roads. Cars can only move on desert and roads. No cell can have more than one unit on it. Water cells recharge adjacent warriors with water, stations recharge adjacent cars with fuel, and walls are just impassable obstacles with no other purpose.

The game lasts 500 rounds, numbered from 0 to 499. The units of player x are only allowed to move at the rounds that are congruent to x modulo 4. For instance, player 1 can only move at rounds 1, 5, 9, etc. There is one exception: cars that have fuel and are located on a road can move regardless of the round number. Additionally, every unit can move at most once per round.

At the end of each round, and for every city, if one team has strictly more warriors inside that city than any other team, then that team conquers the city (unless the city was already owned by that team, when ownership does not change). A city is a (horizontally and vertically) connected component of cells of type City.

Units can try to move to any adjancent cell, either horizontally, vertically, or diagonally, as long as the kind of the adjacent cell is compatible with the kind of the unit. If the adjacent cell is empty, the unit moves there. Otherwise, some rules apply (see below). Units cannot move outside the board.

Initially, warriors have 40 units of food and also 40 units of water, and cars have 100 units of fuel. Every round when they are allowed to move, warriors lose 1 unit of food and 1 unit of water, and cars lose 1 unit of fuel. A warrior whose food or water level reaches 0 dies, and a randomly choosen rival team gets another warrior. A car with no fuel does not "die", but becomes slow not only on desert but also on roads. Units can never have more food, water or fuel than its initial amount.

Any warrior on a City cell recharges its food level to 40, irrespective of the team that owns the city. Any warrior adjacent to a Water cell recharges its water level to 40. Any car adjacent to a Station cell recharges its fuel level to 100. Warriors will have to cross at least one road to reach water from a city (or back). Fuel stations will always be located on some crossroad.

A warrior can try to move to a cell where there is another warrior or a car. Similarly, a car can try to move to a cell where there is a warrior or another car. We first explain the rules that apply in the usual case when the two involved teams are different.

A car that moves to a cell with a warrior runs over it and kills (captures) it. Every moment, any dead unit dissapears from the board, and a fresh unit will appear at the next round, as explained below.

If a car moves to a cell with another car, both cars explote and dissapear, and two new cars are given to the other two teams.

A warrior that moves to a cell with a car comitts suicide, and a new warrior is given to the rival team.

Warriors can attack any adjacent warrior by an order to move to the rival's position. In general (when the atackee has at least 6 units of both of them), the attacked warrior will lose 6 units of food and 6 units of water. Moreover, if any of those levels reaches 0, the attackee will get captured by the rival team. Furthermore, half of the food and water stolen (usually 3 units of each) will be recharged to the attacker, never exceeding the limit of 40.

There is one exception to this rule: if two warriors are inside a city and one of them attacks the other, then the thunderdome rule is used: "Two Men Enter, One Man Leaves". After a bloody fight, one of the warriors dies, whereas the winner's health remains unchanged.

The survivor of a thunderdome fight is decided by the board at random, with probabilities proportional to the water levels of the contestants. For instance, if one has 30 units of water and the other one 20 units, then the former will survive with probability 3/5, while the latter will survive with probability 2/5.

Note that units are allowed to attack units of the same team. As a general rule, this will probably be a bad idea, although in some cases it might be useful to sacrifice an own unit in order to save another one.

When the attacker and the atackee belong to the same team, the rules are adapted as follows. If a warrior gets killed, the new warrior will be given to another team. If a car collides with another car, the two new cars will be given to two other different teams. All those new teams will be chosen uniformly at random.

As a result of all the above rules, the total number of warriors and the total number of cars remains constant during the whole game.

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.

The selected movements of every round will be executed using a random order. For instance, if a warrior tries to move to the position of a friend car, and that

car tries to move to an empty position, then if the warrior moves first it will commit suicide, and afterwards the car will move. Otherwise, if the car moves first, both units will move and survive. As another example, if two enemy cars try to run over a warrior of another team, which does not move, the team of the first car to move will capture the warrior, and afterwards both cars will collide and explote.

After all the selected movements of a round are played, the units captured by each team are reborn. Whenever possible, cars will be placed on roads at the edges of the board, and warriors on desert cells also at the edges, always at a reasonable distance of other units. In the rare cases when this cannot be accomplished, cars will be placed on roads not at the edges, and warriors on desert cells not at the edges, also at a reasonable distance of other units. In even more exceptional situations, cars will be placed on any legal cell, and warriors also on any legal cell, but never inside a city.

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.

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

- *nb_players*: Number of teams in the game (4).
- *nb_rounds*: Number of rounds that will be played (500).
- *nb_cities* : Number of cities on the board (8).
- *nb_warriors*: Initial number of warriors per player (20).
- *nb_cars*: Initial number of cars per player (3).
- *warriors_health* : The maximum (and initial) food and water levels of every warrior (40).
- *cars_fuel* : The maximum (and initial) fuel of every car (100).
- *damage*: The amount of damage that a warrior inflicts when attacking (6).
- *rows*: Vertical size of the board (60).
- *cols*: Horizontal size of the board (60).

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. Run

   ```
   make all
   ```

   to build the game and all the players. Note that `Makefile` identifies as a player any file matching `AI*.cc`.
3. 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`.
4. 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

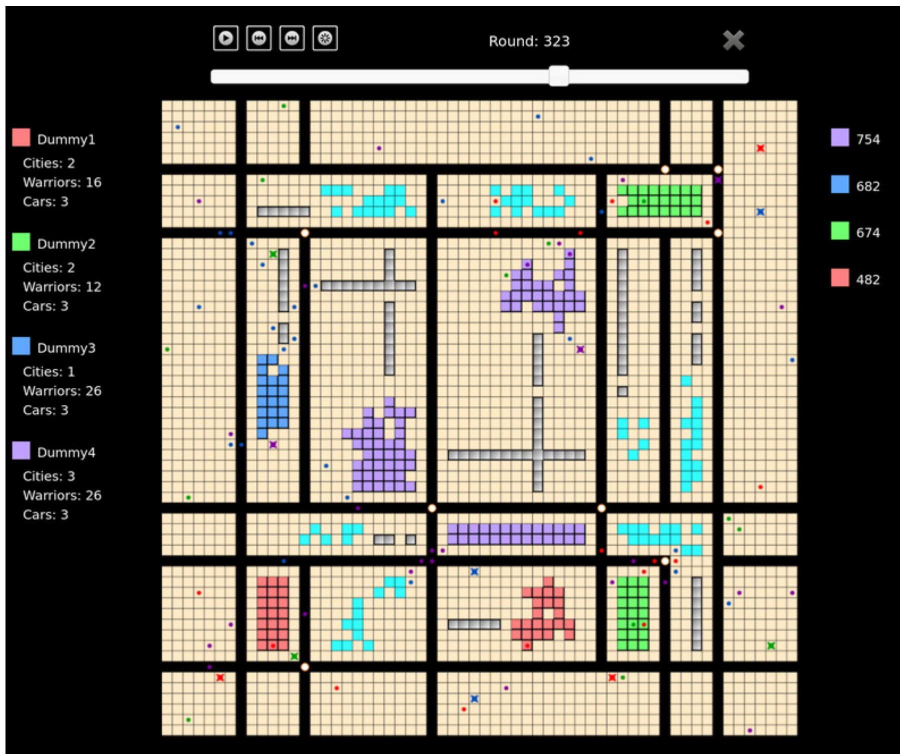
```
./Game --list
```

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

```
#define PLAYER_NAME Null
```

line to

```
#define PLAYER_NAME Rockatansky
```

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 an additional file.

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 Playing against the "Dummy" player

To test your strategy against the "Dummy", we provide `AIDummy.o` object files for this player. 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

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 AIRockatansky.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.

3 Tips

- Read the headers of the classes that you are going to use. Do not worry about the private parts or the implementation.
- Start with simple strategies, easy to code and debug, since this is exactly what you will need at the begining.
- Define simple (but useful) auxiliary methods, and *make sure that they work properly*.
- Before competing with your classmates, focus on defeating the "Dummy".
- Keep a copy of the old versions of your player. Make it fight against its previous versions to measure the improvements.
- As always, compile and test your code often. It is *much* easier to trace a bug when you have only changed few lines of code.
- Use **cerr** to output debug information, and add `asserts` to make sure that your code is doing what it should do. Remember to remove them before uploading your code, to avoid making it slower.
- When debuging a player, remove the **cerr**s that you may have in others players' code, so as to only see the messages that you want.
- If using **cerr** is not enough to debug some of your code, learn how to use `valgrind`, `gdb` or any other debugging tool.
- Make sure that your program is fast enough. The CPU time that you are allowed to use is rather short.
- Try to figure out the strategies of other players watching several games. This way, you can try to react against them, or even imitate or improve them in your own code.
- Do not give your code to anybody. Not even an old version. Not even to your best friend. We use plagiarism detectors to compare your programs, also against submissions to games of previous years.
- You can, however, share the compiled `.o` files.
- You can submit new versions of your program at any time.
- Do not wait until the last minute to submit your player. When there are lots of submisions at the same time, it takes longer for the server to run the games, and it could be too late!

- Do not assume that the code that you last submitted is the one used by our server to play your official matches. Make sure to choose the code that you want to represent you.
- Remember: Keep your code simple, build often, test often. Or regret.