Mad Max 2020

Albert Oliveras

April 17, 2020

1 Game rules

This is a game for four players. Each player commands a group of cars in the dangerous roads of Wasteland. The goal is to get the highest possible score by collecting water bonuses and killing other players' cars. However, gas bonuses also have to be collected in order to keep the cars running and tyres should be avoided to prevent collisions.

The game is played on a bidimensional cylindric universe. After unfolding, this universe can be seen as the result of setting side by side the same rectangle repeatedly along the vertical edge:

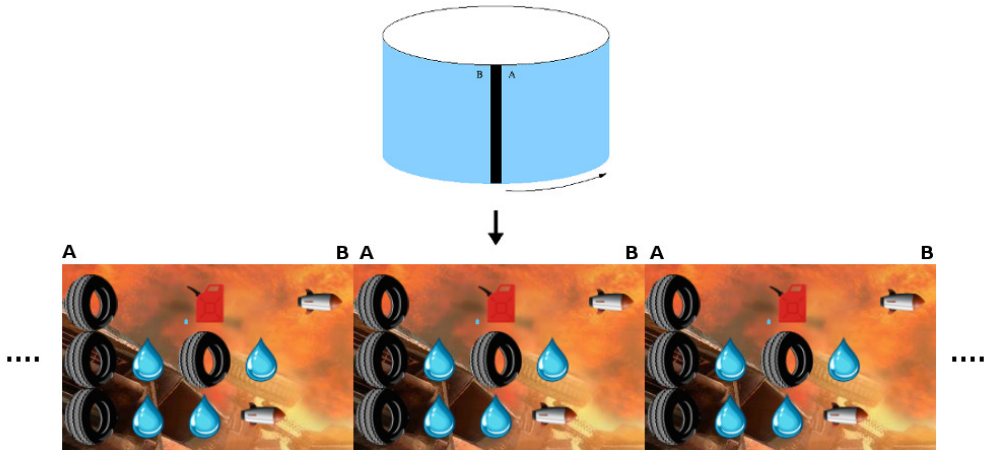

Figure 1: The cylindrical universe after unfolding.

This repeated rectangle is an $n \times m$ grid of cells, where n is the number of rows and m the number of columns. The leftmost upper cell is $(0, 0)$ and the rightmost lower cell is $(n-1, m-1)$. Any pair (i, j) such that $0 \leq i < n$ determines the position of a cell of the universe. Note that two positions (i, j) and (i, j') such that $(j - j') \bmod m = 0$ actually refer to the same cell. In what follows we will refer to i and j as the *row* and the *column* of the position (i, j), respectively.

Each cell of this universe may be empty or may contain:

- a tyre, or
- a car (commanded by one of the players), or
- a missile (previously shot by one of the cars), or
- a water bonus, or
- a gas bonus, or

- a missile bonus.

Players can look up the content of any cell of the universe during the match.

Each player commands a number of cars which is a parameter of the game. Each car has a unique identifying number. The cars commanded by the same player have consecutive identifiers.

1.1 Moving and shooting

At each round a player can command any of its cars to move in a particular direction or to shoot a missile (but not the two at the same time).

In general, the movement of the elements on the universe follows the next rules:

- Cars may move horizontally, vertically or in diagonal. More precisely, in one movement a car may increment its row by -1, 0 or 1, and its column by 0, 1 or 2.
- By default, at each round a car moves horizontally one column, i.e., its row remains the same and its column increases by 1.
- There is a *window* that the cars must remain within. This window is a rectangle of dimensions $n \times m_W$, where $m_W \leq m$, and is dynamic: it moves forward one column per round. Therefore, at round r, its leftmost upper corner is at position $(0, r)$, and its rightmost lower corner is at position $(n-1, r+m_W-1)$. Moreover, a car moving with the default direction remains always within the window.

  If a car is requested to move to a position not within the window, the command will be ignored and the car will move according to the default direction.

  In the game viewer, the window is the part of the universe that will be shown. Since the window is taken as the system of reference for the visualization, it *apparently* does not move; similarly, cars moving by default *apparently* do not move either, etc.
- Tyres and bonuses do not move.
- Missiles move horizontally two columns per round.
- Tyres, bonuses and missiles may be outside the window. Note that in this case, although they cannot be viewed, they still exist on the universe.

A car may shoot a missile if it has at least one in its stock (which is then consumed). There is an initial number of missiles in the stock of each car. This stock can be enlarged by means of missile bonuses. When a car shoots, the new missile automatically moves forward two columns from the car position, and then immediately after, the car also moves with the default direction.

1.2 Collisions in a cell

When a car and a bonus coincide in the same cell, the car consumes the bonus (increasing the number of available missiles if it is a missile bonus, the number of gas units if it is a gas bonus, or getting more score if it is a water bonus). Once a bonus is consumed, if possible it reappears on a random position outside the window. This random position is guaranteed to be previously empty and such that its surrounding square 5×5 is free from missiles or cars.

In the other possible cases of collision, when two elements coincide in the same cell both are destroyed. In particular, if one of the colliding elements is a missile shot by a car of player A and the other element is a car of a different player B, then A increases its score.

When a car is killed, it regenerates if possible after a number of rounds, on a random position of the window. This random position is guaranteed to be previously empty and such that its surrounding square 5×5 is free from missiles, cars or tyres. The number of available missiles is the same as before it died and the gas units are determined by $initial\_gas()$.

As a final consideration regarding collisions, when in a round a car (or a missile) moves from an initial cell to a final cell, in some cases it is considered that it also passes certain intermediate cells. More specifically:

1. When a car moves from cell (i, j) to cell $(i+1, j+1)$, it also passes through cells $(i+1, j)$ and $(i, j+1)$. Similarly when it moves to cell $(i-1, j+1)$.
2. When a car (or a missile) moves from cell (i, j) to cell $(i, j+2)$, it also passes through cell $(i, j+1)$.
3. When a car moves from cell (i, j) to cell $(i+1, j+2)$, it also passes through cells $(i, j+1)$, $(i+1, j+1)$, $(i+1, j)$ and $(i, j+2)$. Similarly when it moves to $(i-1, j+2)$.
4. In the other cases it does not pass through intermediate cells.

The exact order in which intermediate cells are visited can be looked up in the map `dir2all` defined in `Utils.cc`.

1.3 Order of execution of instructions

After the instructions of all players are collected, the following actions take place:

1. First missiles already shot are moved according to their rules.
2. Next a random order is determined among the players, and the instructions for their cars are executed following this order. Instructions given by the same player are executed in their original order. Invalid instructions (for example, moving outside the window limits) are ignored. If a car receives more than one instruction, only the first one will be taken

into account. Cars that have not received any instruction will move by default, in increasing order of identifier.

3. After all instructions are executed, all cars that are alive have their gas units decremented by one. After that, any car with 0 gas units is killed.

4. If appropriate, dead cars and new bonuses (in this order) are regenerated.

1.4 Game parameters

A game is determined by following set of parameters. All of them are specified in the input file, but some of them will always have the same value:

- *number_players*(): number of players in the game. Fixed to 4.
- *number_rounds*(): number of rounds that will be played. Fixed to 300.
- *number_rows*(): number of rows of the universe (and of the window). Variable in $[15, 20]$.
- *number_universe_columns*(): number of columns of the universe. Variable in $[60, 100]$.
- *number_window_columns*(): number of columns of the window. Variable in $[30, 40]$.
- *number_cars_per_player* (): number of cars for each player. Fixed to 2.
- *number_cars*(): total number of cars. Fixed to 8.
- *number_rounds_to_regenerate* (): number of rounds to wait before a car can regenerate. Fixed to 30.
- *number_missile_bonuses* (): number of missile bonuses in the game. Variable in $[5, 30]$.
- *number_water_bonuses*(): number of water bonuses in the game. Variable in $[20, 150]$.
- *number_gas_bonuses*(): number of gas bonuses in the game. Variable in $[5, 15]$.
- *bonus_missiles* (): number of extra missiles obtained when consuming a missile bonus. Fixed to 5.
- *bonus_gas*(): number of extra gas units obtained when consuming a gas bonus. Fixed to 30.
- *water_points* (): number of extra points obtained when consuming a water bonus. Fixed to 10.
- *kill_points* (): number of points obtained when killing a car of another player. Fixed to 30.
- *initial_gas* (): number of gas units given to a car when it is regenerated. Fixed to 60.

All these parameters can be accessed by the players during 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 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.

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

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

   If you use any other architecture, choose the right object you will find in the directory.
3. 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`".
4. 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 -s 3424 < 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`". A random seed of 3424 will be used in this run.
5. To watch the match, open the viewer (`viewer.html`) with your browser and load the "`default.res`" file.

Use

```
./Game --help
```

to see the list of parameters that you can use. Particularly useful is

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

to show all the available players.

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

2.2 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.

2.3 Playing against the Dummy player

To test your strategy against the Dummy player, we provide the object file for it. 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.

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

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 (`AIWhatever.cc`).
- Your code cannot use global variables (use attributes in your class instead).
- 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.
- 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, while they are not in your local environment when executing with `./Game`. 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).
- 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

- **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 we will include in the comparison all programs from previous competitions!). 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, 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.
- 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!
- 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.