TP GO : AI

Game of Tron

Submission rules

You must submit :

```
game_of_tron/
    |-- AUTHORS
    |-- README
    |-- src/
        |-- given_files/
            |-- ai
                |-- ai.go
            |-- game/
                |-- game.go
            |-- main/
                |-- main.go
            |-- player/
                |-- player.go
```

Don't forget to check the following rules before submitting :

- The AUTHORS file must follow the usual format : ** login_x$* or ** firstname.name$*, where the character '$' represents a newline.
- You must follow **exactly** the subject, and especially **follow the prototypes** of the different methods.
- No bin or pkg folder in your project.
- **Your code must compile !**

1 How this project works

This project is about the creation of an artificial intelligence on the a famous arcade game : Tron light cycle

Read the *whole* subject before coding anything. It's especially important this week.

1.1 Given Files

A race manager is available on wiki-prog. It will be needed to test your code. We strongly recommend reading this class to understand how it works. Basically, it executes another program and communicates with it through standard input and output (the console). You can use the error output for tests. What appears on it will be ignored.

Overall, **you won't need to understand how the two programs communicate with each other**. The beginning of this project is available wiki-prog, including the code used to communicate with the rest of the game. It can however help with debugging, so here is how the programs communicate :

```
> [Timeout-ms]\n
> [Width] [Height]\n
> [Self-x] [Self-y]\n
> [Player-number]\n
> [Player1-start-x] [Player1-start-y]\n
> ...

> [Player-number]\n
> [Player1-x] [Player1-y]\n
> ...
< [Command]\n
```

An example on a 20x70 map, with the player we control on (10, 10) and the opponent on (10, 40), with a 100ms timeout :

```
> 100\n
> 20 70\n
> 10 10\n
> 2\n
> 10 10\n
> 10 40\n
> 2\n
> 10 10\n
> 10 40\n
< R\n
...
```

To use the given program, you must run it through a console by giving it your AI as a parameter. One parameter makes the AI play against itself, two parameters make them play against each other. To play yourself against your AI, use a - as a parameter. For example, in a shell :

```
./tron ./login_x/game_of_tron/myAI -
        # Runs the game with your AI as P1 and yourself
          (arrow keys) as P2

./tron./login_x/game_of_tron/myAI
        # Runs the game with your AI against itself
```

To play against your opponent's AI, you can and should exchange binaries. Obviously, do not give your code to another student.

1.2 Ranking

An (almost) real-time ranking will be available on the ACDC's intrachievment [1]. To appear in this ranking, you just have to submit your code and beat the basic AI [2]. All battles will be one vs. one, you will only have **one** opponent.

During a battle between two AI, you will be tested with a set timeout, and in both ways (each AI will be both P1 and P2). If there is a draw, you will fight again with a reduced timeout. Basically, if two AIs are roughly equal, their capacity to adapt to a shorter timeout will make the difference.

The ranking itself will be based on an Elo [3] system. It will be slightly inaccurate and can evolve over time. A reset will be done every day.

1.3 Your Grade

A *small* part of your grade will be determined by your ranking. You will be mostly graded on the mandatory part, although it is not enough for a perfect grade. Some bonuses are required for that.

1. pirates.acdc.epita.it
2. the one given in the base project, which only avoids walls
3. https ://en.wikipedia.org/wiki/Elo_rating_system

2 Implementation of Your AI

Your code will be written mostly in ai.go. The other given files need *not* to be modified, though you may change them if it helps you. You are free to add as many files as you want.

2.1 Mandatory Part : Minimax

You *must* implement all the algorithms described in this part. Your grade depends mainly on these functionalities. However, don't stop here : bonuses are required to get a perfect grade.

You don't *have* to use this section in your final AI, but it must appear in your code, and not without comments.

2.1.1 Algorithm description

Minimax will be the core of your program. It is a simple algorithm, almost hard-coded in its simplest form, that describes an AI for a deterministic turn by turn game. The principle : choosing the action that gives the greatest advantage, while assuming the opponent(s) will do the same. It is strongly recursive : each move will be evaluated using the scores of the moves that result from it. A minimax is very close to a tree traversal : we will often use the tree vocabulary in this subject. In this case, the tree nodes represent the **game states**, and its sons represent the **actions that result from it.**

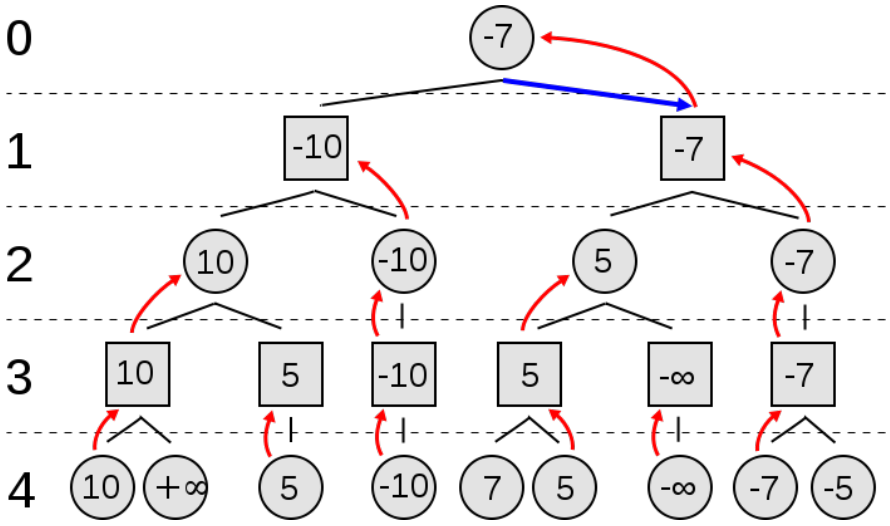
Your method returns the score of the current state, as well as the best action we can take. If the game is over, return directly the score : $+\infty$ for a win, $-\infty$ for a loss. The action in this case does not matter. For each step, you will save the whole map to be able to go back to it, to test states without losing data. For each possible action, you will apply the action to the map, and run a recursive call by specifying that the player has changed. You will keep the best score of the recursive calls : the highest one when you play, the lowest one when your opponent plays.

It is still very abstract and can be difficult to understand. Have no fear. It is a simple algorithm compared to what you have already done. Let's check some pseudo-code to understand it. Be careful, it is not complete and will need to be modified in a later stage.

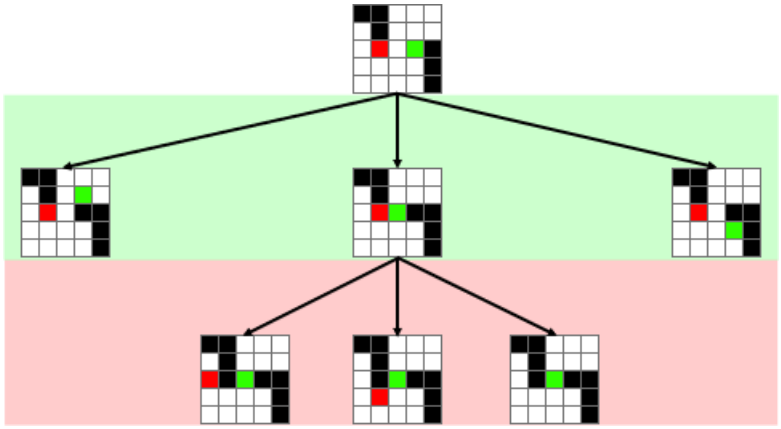
Algorithm 1: Minimax

```
1  function Minimax (maximizing);
   Input  : maximizing whether we maximize this player's score
   Output: Best possible score
2  begin
3      if end then
4          return +∞ or −∞;
5      else
6          if maximizing then
7              bestValue := −∞;
8              forall action do
9                  PerformAction;
10                 bestValue := Max(bestValue, Minimax(false));
11             end
12         else
13             bestValue := +∞;
14             forall action do
15                 PerformAction;
16                 bestValue := Min(bestValue, Minimax(true));
17             end
18         end
19         return bestValue;
20     end
21 end
```

Another way to visualize it, using a tree. Circles represent the player running the algorithm (max), squares represent the opponent (min). All the tests are made after the recursive calls, so read the tree from the bottom to the top.



Another tree example, this time shown on a Tron game. It's important to understand precisely how the algorithm works before coding it. (If the colours don't render well on paper, look at the subject on SOMEWHERE.)



You must follow this prototype. You will understand all arguments by reading all of the subject, which you will obviously do before coding anything (right ?).

```
1 private Tuple<float, Map.Action>
2 Minimax(Map map, int maxDepth, float alpha, float beta, bool ownTurn);
```

You should factorize your code as much as you can. Try not to just copy paste your code from `then` to `else` : you may end up with a 200 lines function after adding some bonuses. Try to write only once lines that appear several times in the algorithm.

2.1.2 Evaluation Function

The basic minimax as we have explained it is cool but it is much too slow. You may have noticed the exponential complexity : we cannot test all possibilities like that. You have to limit the traversal depth, and use a heuristic [4] to evaluate the value of a game state.

Your heuristic will be the method `private float Evaluation()`. You are entirely free regarding the content of this method, but you have to return a higher value on better states.

Some suggestions on the parameters to evaluate : total accessible areas compared to total free area, players position compared to walls and other players, etc. Your method must be fast. Don't limit yourself to your own player, use other players as well. You will be tested by comparing the value of different extreme situations.

4. a heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. [5]
5. Thanks Wikipedia

2.1.3 Alpha-Beta Pruning

Now we start turning our Minimax into a slightly intelligent algorithm, instead of performing a brutal search. Alpha-beta pruning is a modification of minimax, which aims at avoiding evaluating useless branches.

When can we say that a branch isn't worth being evaluated ? Let's take an example. We start by evaluating the branch associated to the `up` move. We find out that it's neutral, not really good, not really bad. Now we continue testing other moves : `down`. Recursive calls, we start by checking the opponent's first move. We find out that he can instantly make us lose. We can already say that we will never go down. In all cases it is strictly worse than going up. And yet we have not evaluated all the opponent's actions. Do we need to ? No. It would only tell us by how much it is worse that going up, but we will not use this information.

From an algorithmic point of view, what does it mean ? The two players have an extreme value, the best action we know of that they can reach. It represents the limit not to exceed. Let's call these values, in a perfectly arbitrary way, α for the maximizing player, and β for the minimizing one. If a state's value goes outside of these bounds, we don't need to evaluate more sons.

In your code, you must use the parameters `alpha` and `beta` in the minimax prototype. Your results must not change *at all*, only your execution time will change (going faster, hopefully).

2.1.4 Iterative Deepening

As you must know, your program's `timeout` isn't fixed. We will even call your program with different values if we wish to. But then, how to make sure we don't exceed the timeout, while making use of the given time ? By using a small amount of recursion ? By limiting the recursion with a division on the timeout ? What if we run our tests on a horribly slow computer ?

One of the solutions is *iterative deepening*. This method may seem inefficient at first, but it actually solves this problem. You try by running your minimax with a very small depth, let's say 1. Did it work ? Good, save the result and try again with 2. It still works ? Awesome ! Let's replace the old result. Eventually we will run out of time : we stop everything, and return the last result we found. We end up doing the same operations several times, but we make sure we use *all* the available time without exceeding the timeout. Some optimizations can still be done. See the following two parts.

Now you must find a good way to stop your minimax without risking exceeding the timeout.

2.1.5 Parallelism

Beware, this bonus is harder to handle than it may seem. But the benefits are huge.

Minimax has an interesting property : the result of each branch does not depend on the other ones (aside from α-β, we'll come back to this later). This opens an interesting opportunity : we can check each branch at the same time. To do so, you can use threads.

You must be careful when using them : if you have several read or write at the same time on the same resource, you can end up with unexpected results. Resource means here shared variables, files, or similar concepts. For example, let's say we keep a maximum value shared between threads. A first thread compares the maximum with its value, sees that the new value is bigger than the old maximum, and write it to the max variable. Between the test and the writing, another thread has written its own max value in there. This value is written over, regardless of its value.

You can use `Mutexes`[6] to avoid this problem. With it you can block an object. The code in the following block will only be run if the object isn't busy, while marking it as busy.

However, be careful about the number of threads : if you start threads on each recursive call, you will end up with thousands of threads. You don't want that to happen : creating a thread and switching from one to another are very expensive operations. We *impose* an (arbitrary) limit of 16 threads, using more leads to an instant disqualification. There is also the problem of the α-β pruning, which is incompatible with parallelism. Clean solutions exist[7], but for now, just create threads on shallow levels of recursion, levels that won't have any pruning.

2.2 Possible Improvements

From here on, you can try to improve your AI. You can use several methods. We have made a little selection of interesting leads to follow. You can, of course, make your own improvements, but *then you must describe them in your README*. Describe your bonuses in your README anyway, but be especially descriptive if it is not in this list. We strongly recommend reading the *entire* section, even if you don't plan to implement all of it, just because these algorithms are interesting.

Note that the algorithms and method described here don't necessarily improve your AI's ranking a great deal. We have chosen methods we think are interesting. Bear in mind, however, that they are not all perfectly fit for this problem, so choose well.

2.2.1 Order of the branches to evaluate

In a Tron, you will evaluate four branches at each node. It's small, but that doesn't mean we should evaluate them in any order. Remember the α-β pruning ? "I am not evaluating this branch, I have already found better." How this can be improved ? You guessed it, by evaluating the best branches first.

To do so, a new opportunity to code fun little heuristics. You can now try and test what works. The goal is to be faster, so always favour speed over accuracy in your heuristic.

2.2.2 Reusing Previous Results

In general, during each turn, both players do what your minimax has calculated as the best moves. You have already done a small traversal on the branches that follow these two moves, which means that you have some idea of what follows. You can reuse this, especially with the order of the actions you can evaluate. (See the previous section.) You can also improve your iterative deepening in the same way. (See the other previous section.)

In a more complex game like chess, we could end up evaluating the same branch several times through different paths. There are very cool ways to keep track of just enough data about visited states to avoid visiting one several times, relying on hash. It's useless here as we can't really reach the same state through different paths, but it's still interesting.

6. https ://gobyexample.com/mutexes
7. "Young brother wait"

2.2.3 Memory optimization

You have probably coded your minimax by copying all of the map and all the players between each action, to be able to go back.

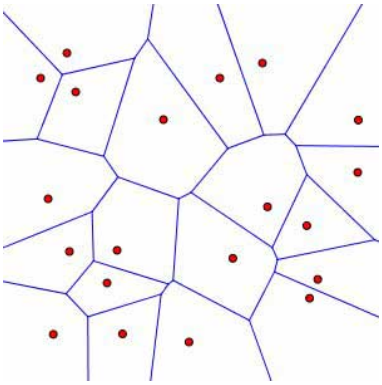
It's slow. It's ugly.

You can do better. You can for example save between each action, not all of the map, but only a list of actions to perform to go back. Less copy, more speed, less memory used, the code is efficient and ecological.

If you want to have some more fun, you can do it using a very elegant way with objects.

2.2.4 Vornoï diagram

A Vornoï diagram [8] is the partitioning of a plane in different regions based on points called seeds. Each seed is surrounded by a region representing all points closer to this specific seeds than any other. It is not an easy concept to explain through words, so here is an example :



In our case, these regions can represent the areas controlled by each player. It can lead to efficient evaluation functions. It can be made easier and more efficient by assuming there are only two players.

2.2.5 More Than Two Players

This seems quite explicit. It won't improve your ranking in any way, but it's still interesting. You will only have to make minor modifications on minimax, especially on the boolean `OwnTurn` (which, as you may have guessed, will be a little less boolean). Each player other than yourself will aim for minimizing the evaluation function. You can go even further, but this is a good start.

2.2.6 Don't Even Try That

Neural network. OK, now we get to the serious stuff. So I say it again : *don't do it*, unless you have done everything else and you have a good amount of free time ahead. You will most likely lack time to do it, but at the very least, you will have learned a lot. If you want to try this kind of algorithm, you likely don't need our help any more, so I'll just link the Wikipedia page [9]. You can also find good information on Google. Go check this even if you don't want to code it, it's fascinating.

In a minimax-based AI, a neural network is used to improve the evaluation function.

Now, it's your turn. Impress us, we look forward to being beaten by our own students.

Can I tell you a secret ? It's been three days since I slept.

8. https ://en.wikipedia.org/wiki/Voronoi_diagram
9. https ://en.wikipedia.org/wiki/Artificial_neural_network