Techniques of artificial intelligence

PROJ-H418

Project report : *Monte-Carlo* tree-search for Checkers

Sami ABDUL SATER
Alexandre FLACHS
Diego RUBAS
Jeanne SZPIRER

Academic year 2021-2022

Contents

1 Introduction : *Monte-Carlo* tree-search

Tree search is an intuitive way to solve a game with a limited number of possible moves. A *Monte-Carlo* tree-search (MCTS) is a tree-search algorithm that exploits **randomness** and **evaluation of simulated games** to decide the next move. The tree is built according to a policy that we hereby define.

Repeat n_{iter} times :

1. **Selection** of the **best** node according to policy

   ▶ **Expansion** of nodes if needed

2. **Simulation** of the rest of the game, starting from the selected node. This simulation ends with a **reward** that takes into account if the game has been won or not.
3. This reward is **backpropagated** to the selected node.

Once all the simulations have been done, the tree is considered to be computed (though not necessarily fully expanded) : we then select the **best child**.

1.1 Parameters

Are variable :

- The selection policy
- The best-child selection policy
- The number of iterations

1.2 Optimization and constraints

There are no particular mathematical constraints to ensure for this project. However, constraints are to be imposed to make it sure it runs in a **realistic time**, e.g. 15 seconds by move.

Under this time, the parameters of the search (n_{iter}, the policies, and more) must be tuned to **optimize the win rate**.

This report presents the implementation of a MCTS on top of a Checkers game. Explaining first the rules, very briefly, we then explain the implementation itself before presenting results of our AI agains a **deterministic** AI (minimax).

1.3 Our contribution

We took the implementation of a Checkers game with a minimax AI on top of it from an Open Source repository. Implementing MCTS required a huge refactor, at the game level and thus also at the minimax level. After implementing MCTS and refactoring, a benchmark was run for different parameters, which lead to an analysis of the behaviour of the AI, and the influence of the parameters on the execution time and the win rate. After this, the work concludes on some further improvements that were conducted and that can still be conducted.

2 Rules of the Checkers game

Let's briefly go through the rules of Checkers game. Particular terms will be used and highlighted, that will be important for the algorithm.

The game opposes two adversary, here named `RED` and `WHITE`, and consists of a **board** and **pieces** on it, each belonging to one player. Each piece then has a color, and the board has pieces on it. Here are some additional information :

- A board can call a function to get all the pieces of a certain color
- It is possible to move a piece of the board using a `move` function
- A piece has a defined **position** (x, y) where x denotes the row and y the column of the piece. A piece is hence defined by a color and its position : $P = (C, x, y)$

2.1 Beginning of the game

Each player has 12 pieces, that begin at the same position for every game, and the starting position is the conventional position for Checkers game. From here, the `WHITE` begins (by convention) and can perform a move.

2.2 Movements

In this section, we define with words how a player can move a piece. We could define it in an algorithmic way, but this wouldn't be particularly relevant for the sake of this report.

A player can only **move** a piece in diagonal, going forward, and can move only one row forward, unless an ennemy piece is on its way. In this case, if the piece can reach a place on the board and some enemy pieces are on its way, the enemy pieces are discarded and the initial piece can find its final destination. We say that the pieces has **skipped** n pieces if n enemy pieces were discarded.

If a piece reaches the opposite side of the board, it becomes a **queen** and can from now on move backwards.

2.3 Endgame

A game ends when

- a player has no pieces left : the adversary wins ;
- all remaining pieces are queens and no piece was discarded in the last 20 moves : it ends as a draw.

3 Implementation of MCTS to Checkers

To implement the tree search, we need to define a tree and the policies associated to the search. Each time the AI has to play, it calls the algorithm, beginning to build the tree (as described in the introduction). Once the tree is built, it selects the best move according to a policy that will be described further.

3.1 Nodes

A node in the tree corresponds to

- The parent node
- A **state** of the game : a `board` element
- The move that lead from previous node to this one (*parent action*) : a `move` element
- `visits` : number of times that this node was visited during the search
- **reward** : number of times that this node led to victory

When the AI is instanciated, the root node has no parent and no parent action, `visits` is set to 1 and `reward` is set to 0.

The resulting constructor for the class `MCNode` can be found on listing 1.

3.2 Selection policy

To select a child node from which perform a simulation, the current node first needs to check if it has children, and if so, if all children have been explored. That is, the current node builds a list of possible children (resulting from the possible moves) and looks for children that are not currently in the tree.

Thus, if the node is currently not **fully explored**, we **expand** the current node. If the node is fully explored, we select the **best child** to perform the simulation.

Intuitively, if there is a sequence of fully explored nodes that leads to a leaf, this will be the privileged path. Otherwise, the algorithm will return the first created child node of a non-fully-explored node. This yields in listing 1.

3.3 Expansion

When a node N needs to be extended, first a list of possible moves is created. Then, a random move r is drawn and a node N_r is created from this move, having N as **parent node** and r as **parent action**. This yields in listing 2.

3.4 Best child policy

There are multiple calls to the best-child policy :

- When the tree is built and we need to perform an actual choice : we choose the best child node of the root node
- During the selection, when a node is fully explored and we need to go down a level to look for a leaf node to select or a node to expand.

During the exploration of the tree, the attributes `reward` and `visits` of each node are updated, such that at any time, it is possible to define, for each node, a value evaluating the node. We chose the following values[1] :

$$\begin{aligned}
&\blacktriangleright \text{ Exploration} \quad d(N) &&= \frac{\texttt{N.reward}}{\texttt{N.visits}} \\
&\blacktriangleright \text{ Exploitation} \quad e(N) &&= \sqrt{\frac{\log_2 \texttt{N.visits}}{\texttt{N.visits}}}\ , \\
&\blacktriangleright \text{ Score} \quad s(N, w_e) &&= d(N) + e(N) \cdot w_e
\end{aligned}$$

where w_e is a parameter to define, to make the famous trade-off between exploitation and exploration.

Once this score computed for each child node, we select the one with the best score (or a random one among the equivalent children).

3.5 Simulation

Once a node is selected, we can simulate a game until a final position is reached. The game simulation is done by playing random move after random move. Our first implementation contained heuristics to choose "better" moves for the simulation, but this resulted in a huge increase in the execution time. A first lesson for the implementation of MCTS hence was : let `python.random` do its thing.

[1]Insérer source wikipédia

4 Benchmark and analysis of the AI

4.1 Initial parameters and approach

Once the algorithm is written, some initial parameters must be choosen. Here : the number of iterations, that we will note n, and the parameter of trade-off between explotation and exploration, that we will note p. As the performance of the AI will depend on those, an initial choice was made, arbitrarily, and depending on the behaviour and performance of the AI, they have been modified.

The most important conclusion that was pointed out was the fact that the iterations must be **as fast as possible**, because the more n can grow, the better. This means that the initial **implementation of the game** must be rethought if it is not optimized for being as fast as possible. Then, during the execution of the algorithm, some **slow methods** must be avoided, e.g. `deepcopy`. Avoid some redundant calls to functions, etc.

With such cleaning of the algorithm, the AI went from 20 seconds by move with $n = 200$ to 20 seconds by move with $n = 20k$. Note that the documentation about MCTS informs us that having relatively small n, like $n = 200$, will result in an AI playing randomly.

n is a parameter that tells how much the tree is "studied" : we want a tree with the highest n possible for a realistic time. On the other hand, p tells us about the way the tree is going to be used.

- Small p : exploration has a privilege

- Big p : explotation has a privilege

Hence, once n was set to being the biggest possible for realistic execution time, p can be tweaked to have a smarter AI : should we explore more, or exploit more ? This will depend on the AI's performance for each p, and will be discussed in the next sections.

4.2 Benchmark setup and implementation

To analyze the performance of the AI, the following tests were conducted. 4 values of n were tested, and 8 values of p. They can be seen on table 1. For each combination, 10 games were simulated, giving a total of $8 \times 4 \times 10 = 320$ simulations, that took about 26 hours to run.

n
5000
10000
15000
20000

p
0.25
0.5
0.75
1
1.25
1.5
1.75
2

Table 1: Test values for n and p.

The implementation can be found in the `benchmark.py` file.

4.3 Wins and losses

From the 320 simulations, we found nothing to conclude on the performance on the AI on its winning ability. There are 6 wins out of those 320 games, and the rest are draws and losses[2]

From this, we conclude that no matter the value of p, the MCTS AI can still not find its path to victory, and that n should therefore be improved, by optimizing the run in the initial implementaion, or finding another way to speed up the process (as will be discussed later).

4.4 Execution time behaviour depending on the parameters

In this section, we answer to the following questions to understand the reaction of the execution time in result to modifying the parameters :

- How does t react when increasing n ? Linearly ? Exponentially ? This is answered on figure 1.
- Does p have an influence on the execution time ? Answered on figure 2.

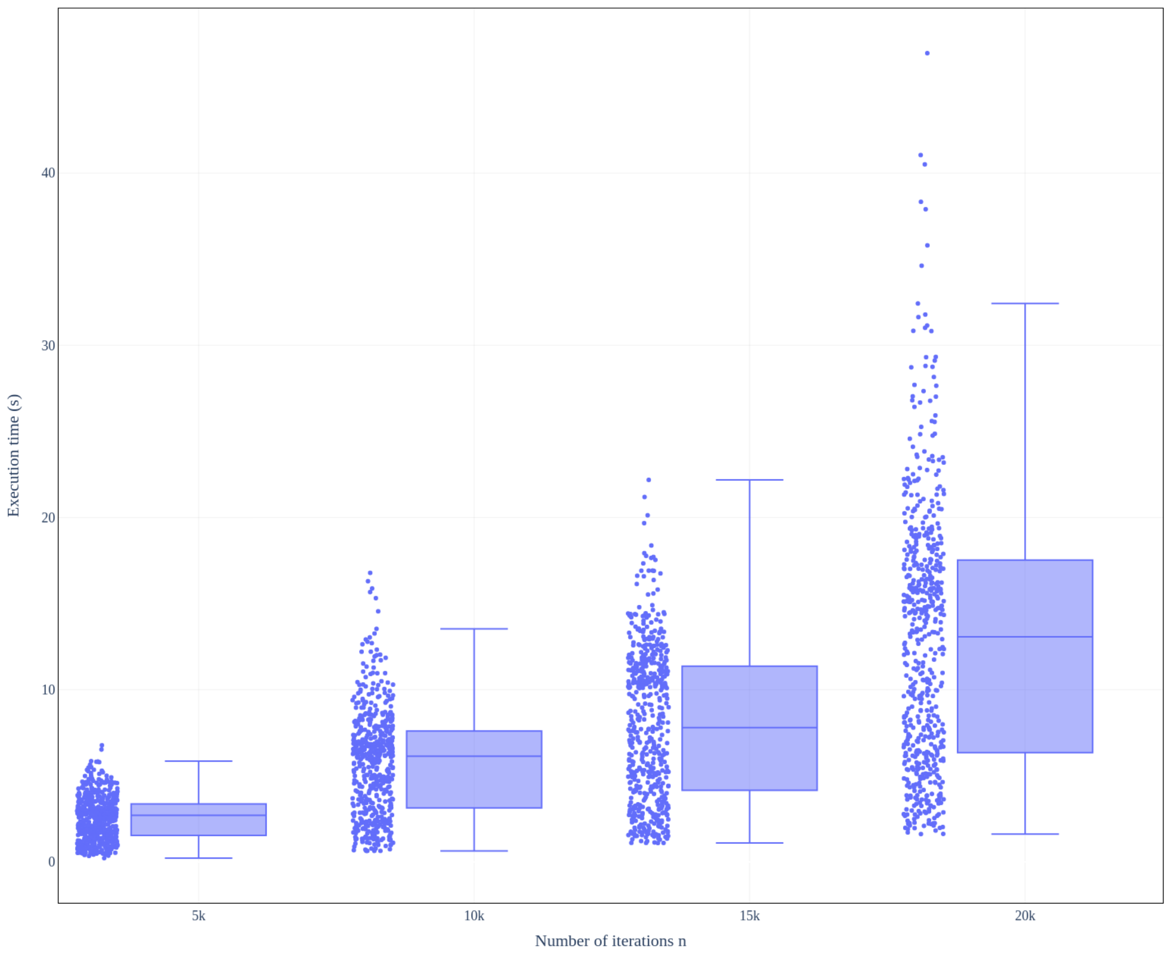

Figure 1: Box-plot of the execution time (in seconds) with several values of n, for $p = 4$. A linear dependance can be seen for the median.

[2]The initial implementation of Minimax thinks that it is winning as long as it has more pieces left, and the game ends up by drawing.

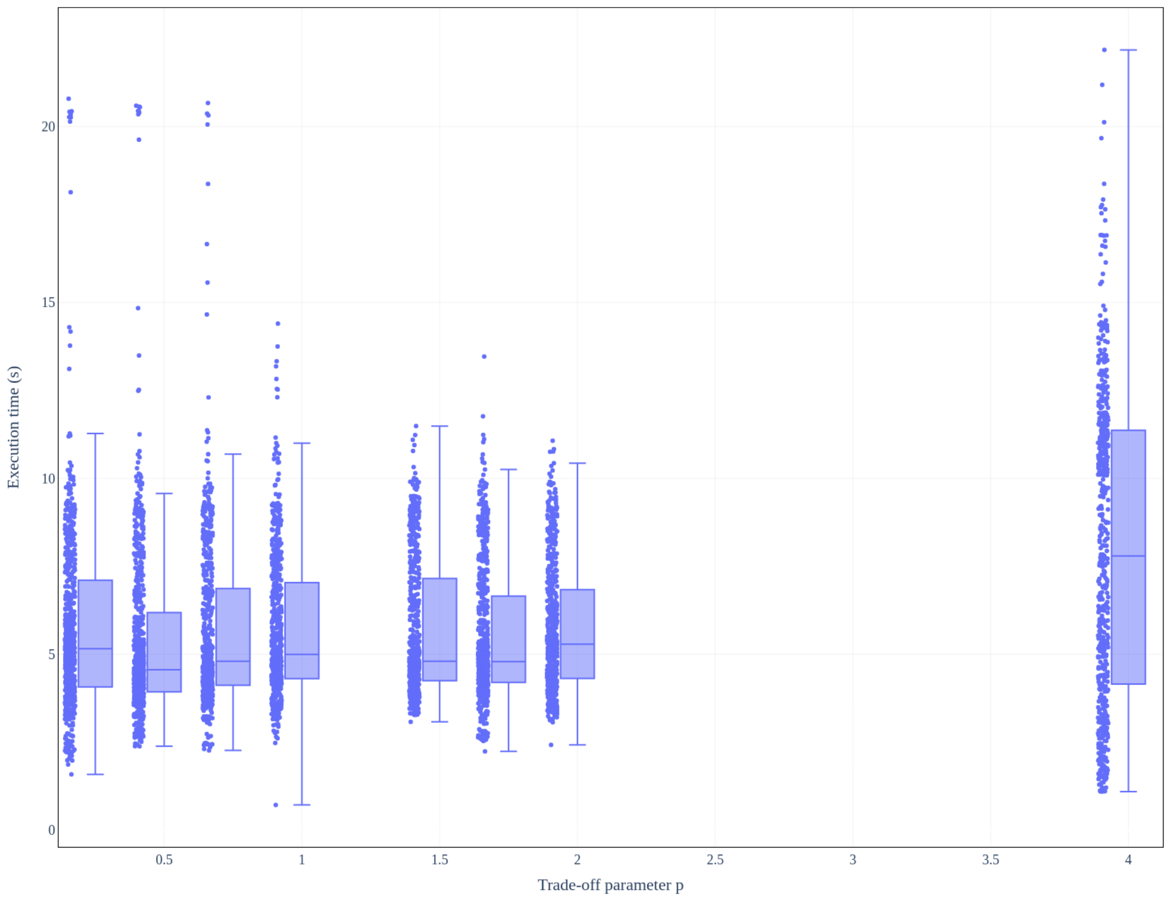

Figure 2: Box-plot of the execution time (in seconds) with several values of p, for $n = 15k$. As expected, the average execution time does not depend on p, hence is constant, although it seems like the execution time seems to spread for high p.

4.5 Evolution of the computation time through games

Tweaking the settings of MCTS or minimax could in theory allow to reach equivalent results as minimax but these would not be achieved using the same computation time. This section explores this idea by comparing the evolution of this time for both algorithms.
As Monte Carlo Tree Search simulates a game to its end at each iteration it is reasonable to think that the computation time of each move will decrease during the game. As the end approaches, the number of moves to simulate decreases. Minimax on the other hand should use a more uniform time during the game since it is a simple tree exploration up to a certain depth (with pruning which will on average offer a constant improvement) It could be interesting to see if the time of MCTS ends up better than the one of minimax, and if so around which turn it happens. This could allow to use both algorithms at their best.
Figure 3 shows this evolution. One can notice that as expected the computation time of MCTS decreases with time, by a factor of almost four between the beginning to the end of the game. Most of this evolution happens in the first twenty-five moves. This peak in the beginning is largely due to the large search space the MCTS has to go through. Indeed, MCTS builds a tree from scratch every time it starts the game. A possible way of mitigating this is using a pre-built tree such that MCTS is better able to exploit possible move rather than explore the search space. This exploration is the same at the beginning of every game and can thus be memorized. This can be achieved through serialization of a tree. However, minimax times are so low (for similar final results) that MCTS never reaches them. Improvements in the implementation could change

this as mentioned in section 4.1.

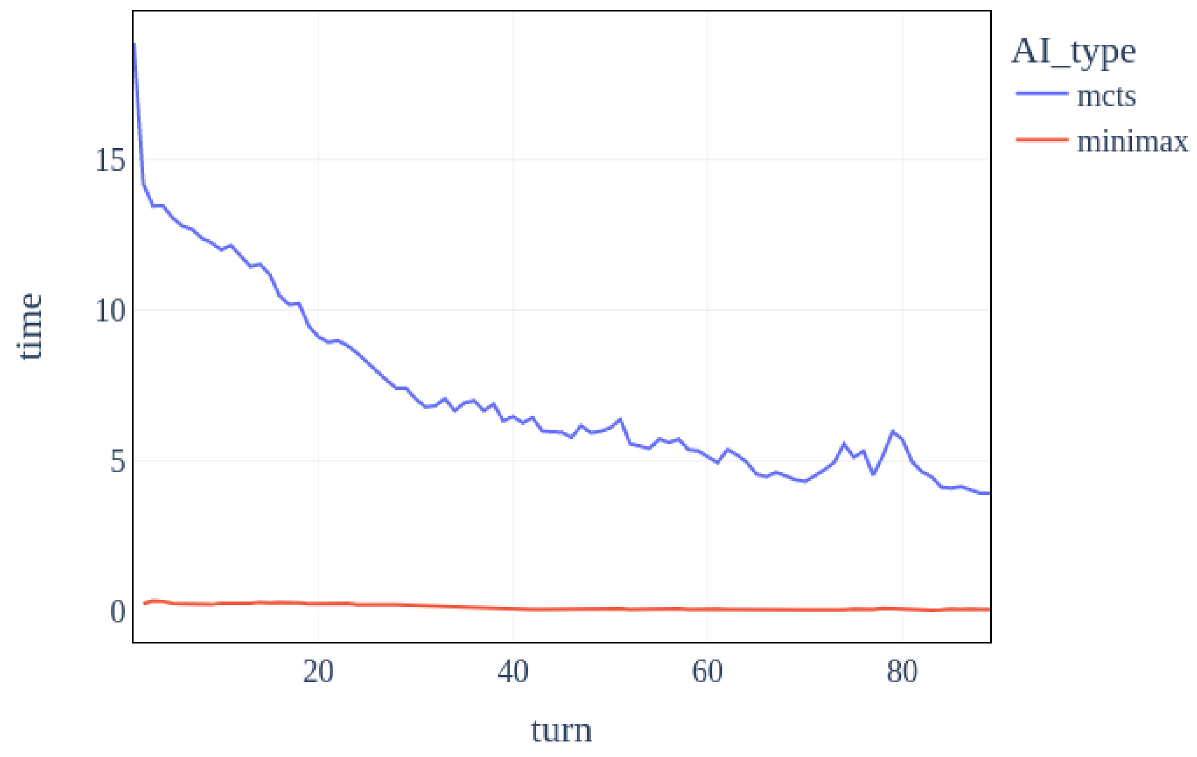

Figure 3: Computing time [s] of both used algorithms (MCTS with $n = 20k$) as a function of the number of turn played. MCTS diminishes but does not reach minimax in these settings. The dataset consists in about 80 games with $n = 20k$ and 8 different parameters of p. However, as p doesn't play a role in the execution time (as shown on figure 2), we can consider that the 80 games are indeed comparable in their execution times.

4.6 Number turns in a time t

In order to continue the exploration of the combination of minimax and MCTS started in the previous section, a histogram was made to compare the number of turns in a certain time for minimax and MCTS. This histogram can be found on the figure 4. It has been made using $n = 20k$, the maximum n that we set for *realistic times*.

The graph presented earlier gives you an idea of what the histogram will look like. Indeed, the minimax rounds always take less time than those made by MCTS. The time taken per minimax is always between 1 and 3 seconds. By the way, on the histogram, some times look negative but this is simply the distribution chosen for the buckets used. The minimax algorithm is much more stable than MCTS, the times are concentrated in the first two buckets of the histogram. For MCTS, the time taken for the rounds changes a lot. A peak is reached between 7 and 9 seconds, i.e. more than 3 times the average time for minimax.
However, the number of laps that take more than 15 seconds is quite low compared to the rest. Even though MCTS takes longer than minimax, this time is still reasonable and considering that this algorithm may be more efficient at the beginning of a game, it would certainly be advantageous to use both algorithms.

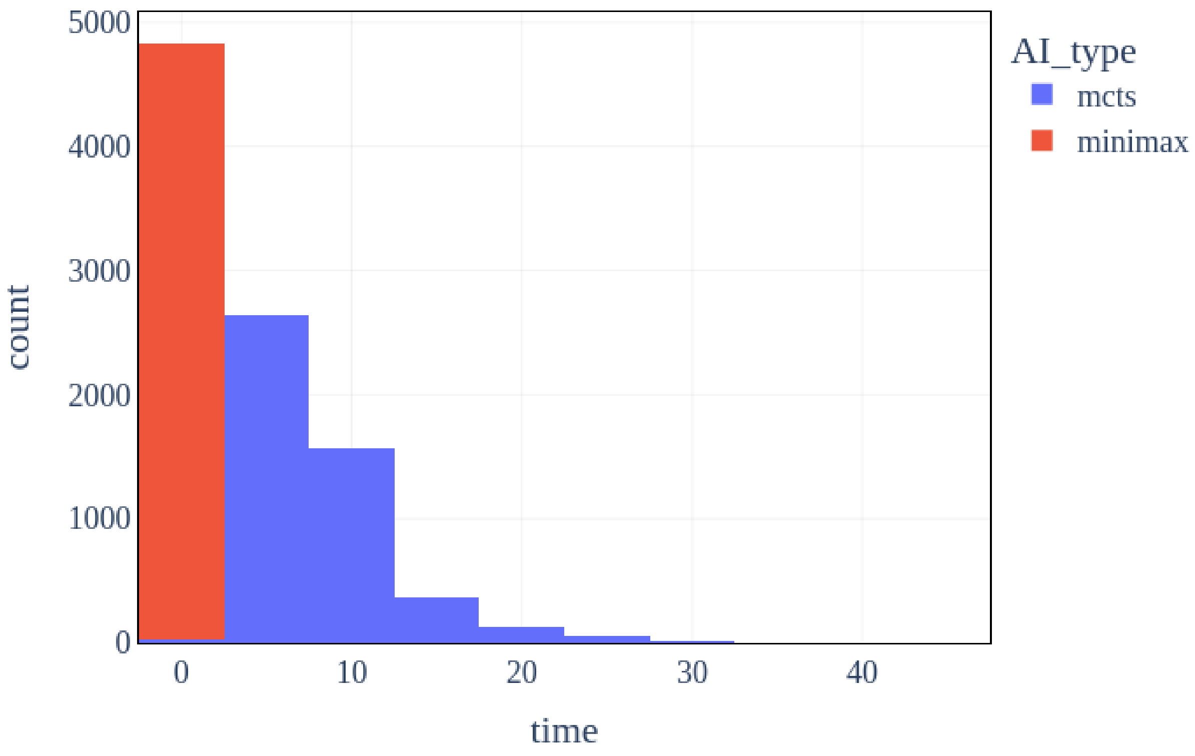

Figure 4: Histogram of number of turns of both used algorithms (MCTS with $n = 20k$) that take a certain time [s]. It is the same dataset as figure 3, and the same argument holds for the coherence of the used data.

4.7 Benchmark conclusion

In conclusion to our benchmark, we can make a global comment on figures 1, 2, 3, and 4.

- First, the documentation about MCTS was clear on how important it is for n to be large. Thus, the main trade-off for humain constraints was on having a large n with realist execution times.

- Figure 2 has shown that the only parameter to regulate the execution time was n. p does not play a role. A prior result was the fact that the code need to be optimized to be able to run more simulations, and we were able to speed-up the algorithm by 10 just by cleaning code.

- Figure 1 has shown that the execution time was linear in n, which is coherent because doubling n mean doubling the number of simulations, so doubling total time.

- Figure 3 has shown that MCTS takes in average a lot of time in the beginning of the game, and the execution time tends to stabilize after. This is a huge difference compared to minimax and led to the understanding that we can not use MCTS like we use minimax, meaning, we can not just implement the algorithm and let it run in a static way.

  A possible use of MCTS would be to implementating it alongside another algorithm that would be, like minimax, stable in the first rounds ; or modify the parameters during the run (dumber moves but low n in the first rounds, smarter moves and higher n in the last rounds, etc.), ...

- However, plot 4 shows that in average, 90% of the moves played my MCTS can be stored in 2 buckets of a histogram, which shows a similar behaviour for each game.

5 Conclusion and further improvements

In conclusion, *our implementation of* MCTS turned out to be unefficient against minimax, even with a small parameter of search. This was confirmed by a very low win rate of 5% on 320 games, and using MCTS's version with the largest execution time while staying realist (15-20 seconds $\Rightarrow n = 20k$). This result led to the major conclusion that with the same execution time, a static version of MCTS is very poor compared to minimax.

Furthermore, an analysis of the average execution times of the AI throughout a game has led to conclusions presented in section 4.7, giving tips on how MCTS could be used more appropriately.

The only objective to maximize was for us the win rate, because we simply thought that correcting our MCTS implementation would be enough to win games and start optimizing the win rate over p. The results were pretty straight forward and the win rate was always near zero. Then, a further improvement can be made on the **benchmark**, not considering only one binary output "win or loss", but starting to take other things into account : how far has the game gone, how many pieces are left, etc., *i.e* **using heuristics** to evaluate a game.

Using heuristics on the simulated games would allow us to rank them according to their performance so see which kind of p gives good performances : high exploration or exploitation ? So, the use of heuristics to improve the benchmark would probably the first thing we would do to get better value of p. Careful : heuristics need to be wisely chosen, analyzed, etc.

Secondly, the implementation of the game could be analysed more in details to clean more code and try to get n higher for realistic times. The use of more efficient arrays structures in Python (use of Numpy, for example) could lead to execution time decrease, etc.

Finally, we could make use of the analysis and make a dynamic AI using different algorithms according to the current state of the game : use minimax at the beginning, MCTS at the end, changing n during the game, etc.

This project allowed us to implement an AI algorithm and see in front of our eyes that things are not as easy as "implementing and see winning". Benchmarking allowed us to understand the behaviour of the AI, imagine how it can be better used, what approaches can be lead to improve it, and even better : what would we be more focused on if we had to go over the same work case.

A Listings

```
 class MCNode:
     def __init__(self, state: Board, color, nb_king_moved, max_it, parent=None, move: Move = None):
         self.state: Board = state
         self.color = color
         self.adv_color = WHITE if self.color == RED else RED
         self.reward = 0
         self.visits = 1  # We always visit newly created node
         self.parent = parent
         self.parent_action = move
         self.children: List[MCNode] = []
         self.children_moves = []
         self.max_it = max_it
         self.nb_king_moved = nb_king_moved  # To keep a track to see if the game needs to end
         return

```

Listing 1: Constructor of a Monte-Carlo Tree-Search Node

Algorithm 1 Selection of a node. Calls the **Best Child policy** and the **Expansion** procedure.

```
procedure SELECT($N$)
    $N$                                         ▷ MC Root Node
    CurrentNode = $N$
    while CurrentNode is not leaf do
        if CurrentNode is not fully explored then
            return CurrentNode.Expand()
        else
            CurrentNode = BestChild(CurrentNode)
        end if
    end while
end procedure
```

2

Algorithm 2 Expansion of a node

```
procedure EXPAND($N$)
    $N$                                         ▷ Current MC Node
    $r$ = random(RemainingMoves(N))
    newBoard = $N$.board.simulateMove($r$)
    $N_r$ = MCNode($N$, $r$, newBoard)
    return $N_r$
end procedure
```

B Genetic Algorithm to tune parameters

This section contained a work that has been done to optimize the trade-off parameter p using a Genetic Algorithm. Unfortunately, as simulations took too much time, this idea has not been used widely. We preferred to sequence our work : implementation, analysis, and then improvement according to the results of the analysis ; but the "improvement" part couldn't be conducted.

B.1 Parameters to find

The implemented genetic algorithm allows to calculate the value of some parameters in order to improve the main Monte-Carlo algorithm which plays checkers. These parameters are :

– the number of iterations

– the safety factor (see Safe heuristic)

– the exploitation factor.

Except for the number of iterations, the factors to be found can be between 0 and 1 and are there to determine the importance of the coded heuristics.

B.2 Algorithm

B.2.1 First generation of parents

The algorithm starts with a random generation of "Villagers" (representing the population). *POP_SIZE* villagers are created and their parameters are randomly drawn.

B.2.2 Simulation of games

For each villager, a thread is created to make him play against Minimax *NB_GAMES* times. This saves a lot of time. Once all the games are finished, the parents can be ranked from best to worst. The quality of a parent is defined by the following fraction,

$$\frac{reward}{nb\_simu}$$

The *reward* is the number of points recovered for all games played and *nb_simu* is the number of games.

B.2.3 Evolution of the population

Once the population has finished playing, merging can begin. *NB_KEEP* parents are kept from the current population (the best) to create the next one. Pairs are then formed and there will be as many pairs as there are individuals missing in order to have the same amount of population at every generation. Each couple formed creates a new child villager through the single point cross-over method.
After merging, one or some children are eventually mutated. The rate of mutations is determined with the global parameter *RATE_MUTATION*. For every new villager, the algorithm picks a random real between 0 and 1, if the real is smaller than the rate, a mutation occurs on the villager. Then a parameter to determine (cf. B.1) is chosen randomly and a mutation is done. If it is the number of iterations, it's just increases until it reaches the maximum. If it is one of the factors, the parameter is transform into a binary number and one bit is switched.

B.2.4 Continuation and end

The loop starts again with the newly generated population. Each parent plays a certain number of games, receives a reward, and is allowed to play a game... The algorithm stops when the convergence criterion is checked.

B.3 Further improvements

Here stops the implementation of our genetic algorithm. As already said, an optimization of the parameters needed to be conducted, but GAs didn't seem to be the way.