



## 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

<b>1</b>	<b>Introduction : <i>Monte-Carlo</i> tree-search</b>	<b>1</b>
1.1	Parameters . . . . .	1
1.2	Optimization and constraints . . . . .	2
1.3	Our contribution . . . . .	2
<b>2</b>	<b>Rules of the Checkers game</b>	<b>2</b>
2.1	Beginning of the game . . . . .	2
2.2	Movements . . . . .	2
2.3	Endgame . . . . .	3
<b>3</b>	<b>Implementation of MCTS to Checkers</b>	<b>3</b>
3.1	Nodes . . . . .	3
3.2	Selection policy . . . . .	3
3.3	Best child policy . . . . .	3
3.4	Summary : pseudo-code . . . . .	3
<b>4</b>	<b>Genetic Algorithm to tune parameters</b>	<b>3</b>
<b>5</b>	<b>Results</b>	<b>3</b>

## 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  $n_{\text{iter}}$  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 ( $n_{\text{iter}}$ , 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 against 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 optimization of the win rate over the parameters of the search.

# 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 ;
- 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.

### 3.2 Selection policy

### 3.3 Best child policy

### 3.4 Summary : pseudo-code

## 4 Genetic Algorithm to tune parameters

## 5 Results