ECOLE
POLYTECHNIQUE
DE BRUXELLES

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 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 ;
- 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[1] 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[2] :

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

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 : introducing heuristics

Once a node is selected, we can simulate a game until a final position is reached. The game simulation is done by playing heuristic-based moves. We defined some heuristics to evaluate the quality of a move, and the possible moves are sorted according to this evaluation.

Hence, the game can be simulated by choosing the best move, generating a new board, and changing the player's turn accordingly.

[1] pour l'instant. a modifier si on change
[2] Insérer source wikipédia

3.5.1 Heuristic 1

3.5.2 Heuristic 2

3.5.3 Heuristic 3

3.5.4 Heuristic 666

4 Genetic Algorithm to tune parameters

5 Results

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

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