PSL Research University

Master 2 IASD - MonteCarlo Search and Games

Applying Monte Carlo methods on Marrakech board game

Authors :
Mehdi Bentaleb
Yassine Abou Hadid

Supervisor :
Tristan CAZENAVE

April 2023

Contents

1 Introduction **2**

2 Marrakech Game **3**
2.1 General idea 3
2.2 Game setup 3
2.3 Game rules 4
2.4 Game modeling 4

3 Monte Carlo Methods **5**
3.1 Random 5
3.2 Flat Monte Carlo 5
3.3 UCB 5
3.4 UCT 5
3.5 Introducing the Score variant 6

4 Experiments **7**
4.1 Setup 7
4.2 Results 7
4.2.1 MCTS strategies vs RANDOM for different playouts 7
4.2.2 Tournament of 100 games with 50 playouts 8
4.2.3 Tournament of 100 games with 300 playouts 9

5 Conclusion **10**

1 Introduction

Marrakech is a worldwlide touristic destination, located in Morocco and known for its vibrant markets, stunning architecture, and rich cultural heritage. It gives its name to a popular family-friendly board game designed by Dominique Ehrhard and published by Gigamic Games. Marrakech is a simple and fun game that is easy to learn, making it a great choice for players of all ages and skill levels.

Monte Carlo Tree Search algorithms are a range of powerful artificial intelligence techniques used to solve complex decision-making problems, and they have shown great promise in a variety of fields, from gaming to robotics to finance. In this project, we will explore the potential of some MCTS strategies on the Marrakech board game, using its intricate gameplay mechanics to demonstrate the power and versatility of these algorithms. By applying MCTS to Marrakech, we hope to gain insights into how these algorithms can be used to solve complex problems and optimize decision-making in a variety of real-world scenarios.

Through this project, we have explored several MCTS algorithms to play the game of Marrakech, such as UCT, UCB and Flat Monte Carlo as well as a variation of these strategies we introduced to solve the game problem.
Our code can be found in this **repository**.

2 Marrakech Game

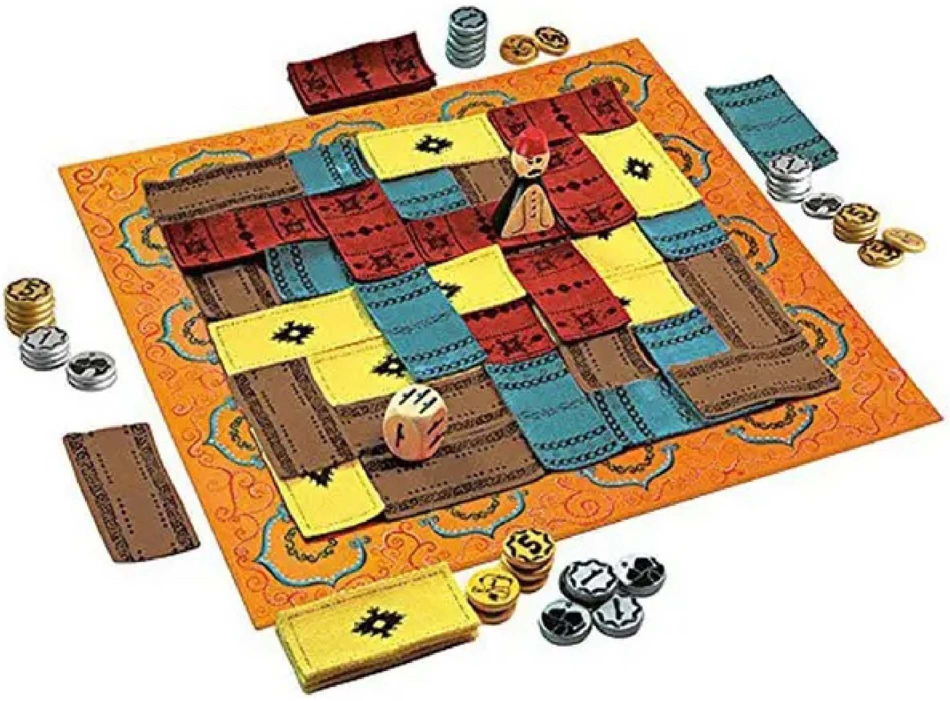

FIGURE 1 – Marrakech game board

2.1 General idea

In Marrakech, the most skilful of merchants will win. In fact, each of the players will try to have the most of their rugs exposed at the end of the game while accumulating the greatest fortune. Through the game, they will take turns moving their merchant figure through the marketplace, placing a rug on the ground each time they stop. The objective of the game is to block and outmaneuver opponents while also collecting the most money. Players earn money based on the number of the rugs they place and the number of tiles they cover. The player with the most money at the end of the game wins.

Marrakech can be played by 2 to 4 players. In our setting, we decided to restrict it to 2 players.

2.2 Game setup

On the 7×7 board, the pawn named Assam is placed in the center of the board facing North. Each one of the two players has 30 coins (30 dirhams), 24 rugs (of 2 different colors). We have also a 6-sided die (with particular values).

It is useful to note that Monte Carlo algorithms are efficient for games with a fairly high branching factor. It it the case here, because at each move, we can choose among $4 \times 49 \times 12 = 2352$ moves (4 orientations, 49=7x7 possible pawn placements, 12 different carpet locations).

2.3 Game rules

In turn, players make the following three moves : they move Assam, if necessary, they pay their opponent, they then lay one of their own rugs :

— **Moving Assam :** The player chooses in which direction they want to move Assam before throwing the dice, Assam can be left alone or turned 90° left or right (he cannot turn 180°).
The player then throws the dice : the number of slippers indicated on the dice determines how many squares Assam is moved. Assam moves in a straight line (not diagonally) in the direction initially selected.
If Assam leaves the market, he follows the about-turn signalled by the arrows (the arrows do not count as a move).

— **Payments Between Merchants :** If Assam ends his move on an opposing player's rug, the player must make a payment to the rug's owner.
The amount owed is equal to the number of squares adjoining the square that Assam has landed on which are covered by rugs of the same color : the player must pay the same amount as the number of squares covered.
The sides of the squares must be touching, it does not count if they only touch diagonally ; Assam's square count is included.
The player makes no payment if Assam ends his move on an empty square or on one of the player's own rugs.

— **Laying Rugs** : The player then lays one of their rugs next to the square where Assam has finished, an edge of the rug must be placed against one of the 4 sides of this square.
A rug can be placed on :
  — two empty squares.
  — an empty square and half a rug (whatever its color) ;
  — two halves of different rugs. An opponent's rug cannot be entirely covered in one go (it can only be covered completely by two rugs).

The game ends once the last rug is laid. Each half of a rug visible and each coin count as one point. The player with the most points wins the game.
In the case of a tie, the player with the most Dirhams wins.

2.4 Game modeling

Before appylying Monte Carlo algorithms, we have created different classes to represent the game :

— **The Board class** : it represents the game board, and therefore the state of the game at time t. It has in particular the most important methods, the one which allows to have the list of the legal movements, the one which allows to calculate the score, and the one which allows to make a random play (playout).

— **The Player class** : it represents a player and allows keeping count of his coins as well as the number of carpets he has left to play.

— **The Position class** : it represents the position of the pawn or half of a carpet.

— **The Move class** : it represents a movement of the game. In particular, it has methods for whether a move is legal or not.

— **The Assam class** : it represents the pawn of the game, which is characterized by its orientation and its position. In particular, the pawn can move and calculate the

number of coins it owes its opponent if it lands on one of his carpets.
— **The Rug class** : it represents a rug. Since it is forbidden to put a carpet on two squares already covered by the same carpet, it is particularly important to keep an id for each carpet in addition to its color to differentiate between two rugs of the same color.

3 Monte Carlo Methods

3.1 Random

We use the random strategy as a baseline. It consists in choosing a random move among the legal moves in each turn.

3.2 Flat Monte Carlo

During a Flat Monte Carlo simulation, the algorithm repeatedly plays out many random playouts of the game from the current state. Each playout involves randomly selecting moves for both players until a terminal state is reached. The outcomes of these playouts are then used to estimate the value of each possible move. The moves with the highest estimated value are then considered the most promising and are selected for further exploration in the game tree.

3.3 UCB

UCB strategy enables to make compromise between the exploration of uncertain branches and the exploitation of most promising branches. Before each playout, we select the move that maximizes a score. At the end, the best move chosen by the AI will be the one that has been played the most times.

3.4 UCT

UCB applied to Trees combines the exploration-exploitation trade-off of MCTS with the Upper Confidence Bounds (UCB) algorithm to balance the exploration of unexplored nodes with the exploitation of promising nodes. It uses a heuristic function to estimate the value of each node in the tree and balance exploration and exploitation using a formula that combines the average reward and uncertainty of the reward. UCT (Upper Confidence Bound applied to Trees) is used to select the action with the highest UCT value. The key advantage of UCB applied to Trees over UCB is its ability to handle more complex decision-making problems with a tree-like structure. Unlike UCB where we start from a initial configuration and the most promising node is determined, the UCT implementation applies the UCB formula at each extended node of the game tree until reaching a leaf.

To implement UCT, we need to keep in memory for each game configuration, the total number of playouts where the configuration appears, a list of the number of playouts and a list of the number of games won after each move from this configuration. In practice, we create a dictionary (Table) indexed by a configuration of the game represented by an integer. This integer is determined by Zobrist hashing, which consists in carrying out the

XOR of the random numbers associated with each cell of the board for the configuration in question. The state of a space is determined by :

1. the presence or absence of the Assam pawn
2. the pawn orientation
3. the result of the dice
4. the coordinate of the cell on the horizontal axis
5. the ordinate of the box
6. the color of the carpet placed on the square, or the absence of a carpet
7. the player during the round
8. the color of the carpet during the round

The random numbers are kept in a transposition table, implemented by a dictionary that we add in the Board class. Each sub-state of a box (from 1 to 6) is itself represented by a dictionary. The two substates 7 and 8 are represented independently of the dictionary.

To calculate the code for a board configuration, all you have to do is XOR it :

— the movement of the pawn
— the placement of the carpet on two squares adjacent to the pawn
— the change of player for the next round
— the change of carpet color associated with the next player

3.5 Introducing the Score variant

In the standard setup described above, the winner is the player who has the most points. To compute the final score, we consider for each player the sum of his pieces and of the squares of the board covered by one of his rugs.

The score of a game s is the score of the first player minus the score of the second one, such as :

— If $s > 0$: player 1 has won with a lead of s
— If $s < 0$: player 2 has won with -s points in advance
— If $s = 0$: there is a tie.

For each strategy (except random), we tried to implement a variant (Score) that takes into account the average score rather than the win rate.

Strategies will therefore favor moves that get the most points in average rather than the ones that win most often.

For this, when in the classic version of an algorithm we would add to the counter of a move 1 for a win and 0 for a loss, we add the final score s of the game instead (which can be positive or negative). The motivation behind this variant is to favour the move that have a better average score between a set of "winning" moves. And and in the same way, favour a move that have a low average score in a set of "losing" moves.

The issue here is that, for some algorithms, it is possible to add the points directly (for example for Flat MC, it is enough to make an average of the scores to evaluate a movement), but for others, like UCB and UCT, it will be necessary to evaluate a node differently, and it would be necessary that the score of a node remains between 0 and 1. In our case, the maximum score is 60 (player 1 has collected all the pieces of player 2) + 5×5 (a point per space of the board covered by one of its carpets), therefore 85 points. We then normalized the scores as follows :

- — First, we clip the score : if the first player wins with a lead of more than 40 points, we bring the score at 40 and if he loses more than 40 points behind, the score is reduced to -40.
- Then we bring the score back between 0 and 1, i.e. we project the segment [-40, 40] into [0,1].

The normalized score is :

$$\textbf{newscore} = \frac{clip(score)-min(score)}{max(score)-min(score)}$$

4 Experiments

4.1 Setup

To have a rational execution time, we did some limitations on the game :

- we have adapted the game for a 5 $\times$ 5 board and 32 rounds (8 rugs per color).
- In order to avoid any randomness in a move, we have chosen to roll the dice before a player's turn. Thus, the player only has to choose, knowing the result of the dice, the orientation of the pawn and where to place his carpet.
- We have implemented 7 strategies : Random, Flat, Flat Score, UCB, UCB Score, UCT and UCT Score

Finally, our strategies demonstration consisted in doing 3 experiments :

1. Playing all implemented the strategies vs Random strategy for 20 games with different values of playouts.
2. Playing a tournament of all strategies for 100 games with 50 playouts.
3. Playing a tournament of all strategies for 100 games with 300 playouts.

4.2 Results

4.2.1 MCTS strategies vs RANDOM for different playouts

Here, we simulated a game of each strategy against random, over 20 games with different number of playouts.

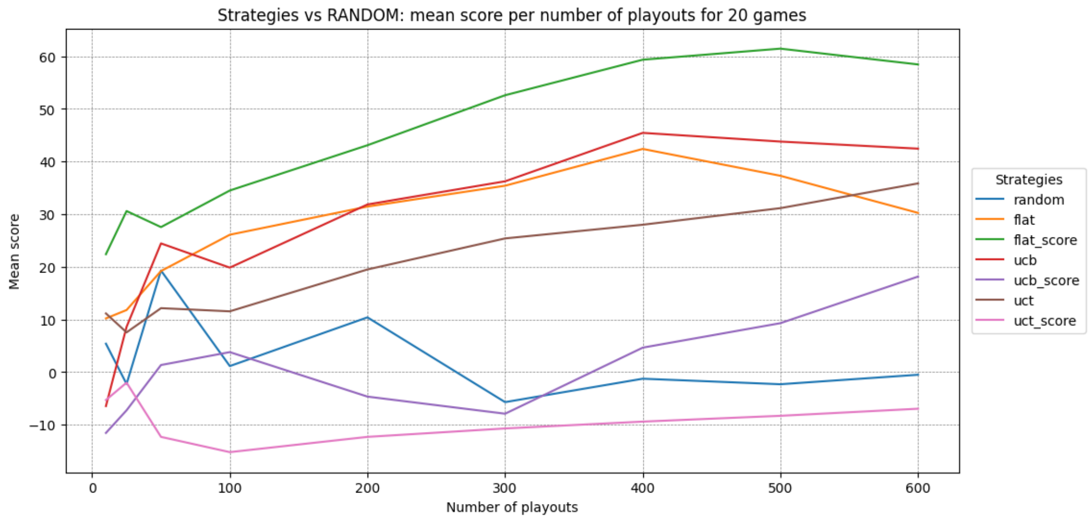

Figure 2 – Strategies vs RANDOM : mean score per number of playouts

We observe that UCT_score makes negative scores against randomness, so probably it loses in average, which is quite strange and possibly indicates an implementation error. Also, we notice that flat_score achieves the best scores means, regardless of the number of playouts, which shows the usefulness of this variant. In addition, we can remark UCB_score seems to improve with the number of playouts, but remain, up to 600 playouts still far behind flat score. This may only indicate that the variant is not at all suitable for UCB. We can also see that UCB is improving as the number of playout increases. Surely with more playouts, UCB would have accomplished even better results.

4.2.2 Tournament of 100 games with 50 playouts

We represent below the results of the tournament between each of our strategies. The tables are read as follows : the row strategy won x% of the time against the column strategy (left plot) with an average score of y (right plot). Recall that average score is positive in case of a victory and negative otherwise

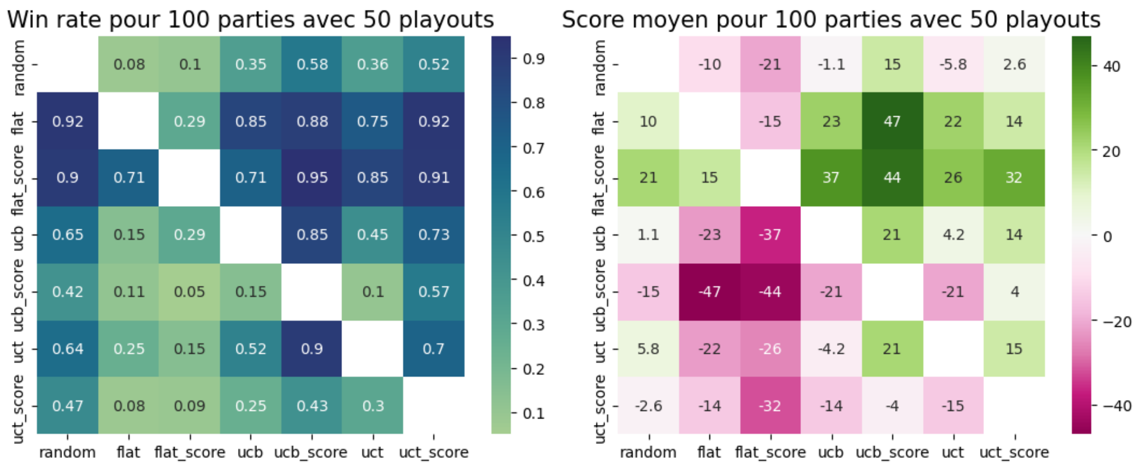

Figure 3 – Results for 100 games with 50 playouts

With only 50 playouts, the flat strategies are much better than the others, as it can be seen on the 2nd and 3rd line, especially against UCB strategies. This can be explained by the fact that over 50 playouts, UCB wasn't able yet to exploit promising moves and therefore has little chance of choosing the best one. We also note that flat_score variant won 71% of the time against classic flat, which seems to confirm that for a small number of playouts, the score variant gives an advantage. Besides, we also note that flat_score actually achieves better average scores than the classic version. On the other hand, the score variant of UCB is not at all efficient. As for the UCT strategy, it loses against flat, flat score and UCB ; it wins very slightly against the random strategy, but largely against the UCB score variant. On the other hand, the UCT score strategy is less efficient than the other strategies. In general : it loses in particular against the random strategy and UCB score. These results are surprising because we expected better performance since the UCT strategy applies the UCB formula at each extended node in the game tree, so theoretically by recursion, the best move is better chosen than in the UCB strategy.

4.2.3 Tournament of 100 games with 300 playouts

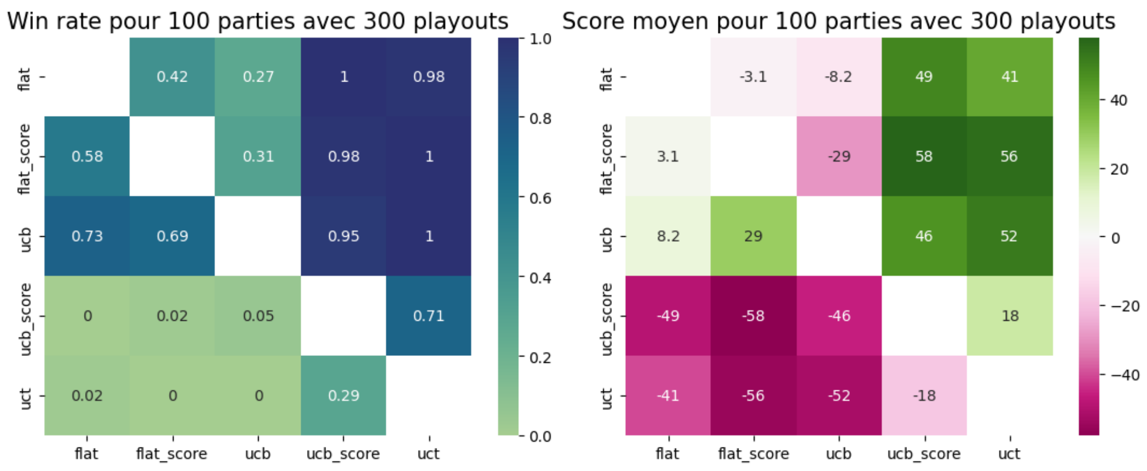

FIGURE 4 – Results for 100 games with 300 playouts

Contrary to the version with 50 playouts, the best strategy is UCB, which has a higher win rate over the other strategies. This shows the interest of evaluating whether a node is promising or not. Also, we can remark that the strategies flat_score and standard flat almost tied, which seems to indicate that the score variant is much less useful for a large number of playouts. However, for games with 100% winrate (flat and flat_score against UCB_score and UCT), the score variant achieves a better average score (58 against UCB score, which is very high, the best possible score being 85). Regarding the UCT strategy, it performs much worse than before : it loses against the other strategies for a large number of playouts, and loses with average scores lower than those observed for the 50-playout version. Again, these are results that cannot be could not be explained. One hypothesis, as mentioned in the first experiment, could be an implementation error.

5 Conclusion

In conclusion, we tried to implement some Monte Carlo Tree Search (MCTS) methods, including Flat Monte Carlo, UCB and on the Marrakech board game, and we saw that they are powerful algorithms for decision-making problems. While the flat method can produce good results in many cases, UCB can potentially provide better results with more plays by balancing exploration and exploitation in the decision-making process. However, the performance of UCB may depend on various factors such as the complexity of the decision-making problem and the number of available plays. Therefore, the choice between the flat method and UCB should be based on the specific problem and the available resources.

One improvement could be trying out other MCTS strategies such RAVE, nested Monte Carlo Search.. We could also try to reimplement the UCT variant strategy since it seems to output erroneous results.

As a side note, we used Google Colab Pro GPUs to perform the heavy computations.

References

[1] Tristan Cazenave, Julien Sentuc, Mathurin Videau (2021), *Cosine Annealing, Mixnet and Swish Activation for Computer Go*, Advances in Computer Games 2021.

[2] ADAM BERENT (2019), *Transposition Table and Zobrist Hashing*, URL : https ://adamberent.com/transposition-table-and-zobrist-hashing/

[3] https ://github.com/sor8sh/Marrakech