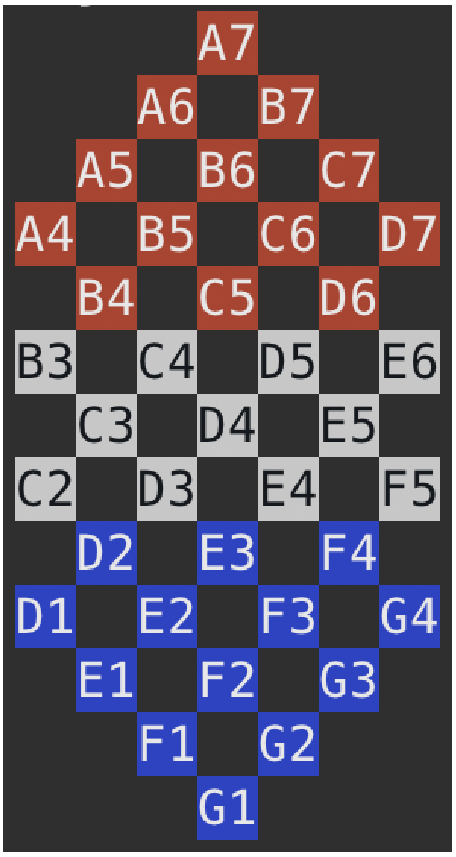
StuckWin - Report



IUT Nord Franche-Comté - Group n°16 – HUMBERT Ewan – SIOUAN Tom – 2022-2023

I. Table of Content

1. In search of a suitable algorithm 4
2. How it works? 5
3. Benefits, limitations and statistical analysis 6

1. In search of a suitable algorithm

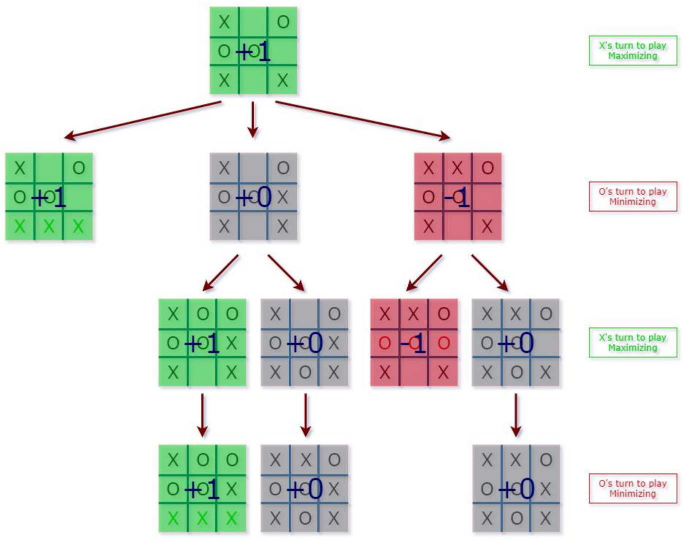
This project consists of creating a game. This latter is composed of 2 teams represented by 13 pieces each (red and blue). The goal is to get stuck before the opponent. By "stuck", we mean when all your 13 pieces can't move on any of the 3 possible destinations (top-left, top-middle, top-right). For the SAE 102 we had to find an algorithm that acts like an artificial intelligence which means finding and playing the best move possible in order to win.

We firstly thought of a "homemade" strategy that consists of moving the furthest back pieces, so that the player's pieces are always grouped and thus more easily stuck.

Then we made some research to find if it exists an already-existing algorithm that would meet our needs.

We discovered the Minimax algorithm. Basically, it browses every move possible for the player, then from each of these moves, browses every move possible the opponent can do, and so on until someone win.

The algorithm then chose the move that results in the lowest score. This is done by assigning a score to each possible game state, with higher scores indicating a better position for the computer and lower scores indicating a better position for the opponent. The computer then chooses the move that maximizes its score and minimizes the opponent's score.



Schema 1: Minimax for the game Tic Tac Toe by Yiğit PIRILDAK

That schema is represented under the form of a tree where each state is a node. I won't explain further because we tried to implement it but found ourselves in an infinite loop.

Instead of trying to fix it, it has been thought that this algorithm becomes cumbersome when the game has many possibilities. Minimax is used a lot for Tic Tac Toe, but the Stuckwin game has many more squares and therefore many more possibilities. It is therefore more time consuming to calculate all the possible states of the game. If we were to represent it by a tree (like in the schema 1 above), the tree would be huge with lots of nodes.

So, we searched for another algorithm that would be more suitable for a game with a certain number of squares.

We finally found THE algorithm: the **Monte Carlo tree search**.

2. How it works?

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that is used to find the next best move in a game. It is called a “tree search” because it builds a search tree as it considers possible moves, and it uses “Monte Carlo” methods because it uses random sampling to estimate the value of each move.

In the following explanation, a node is a state of the game, and its children are all the possible moves the player can do from this state. Then these children are nodes that also have children representing the possible moves the opponent can do from each of these children. And the process repeats until there is no possible move, meaning someone won (these nodes are called “leaf”). This forms a tree with as a root, the current state of the game.

Here's how MCTS works in more detail:

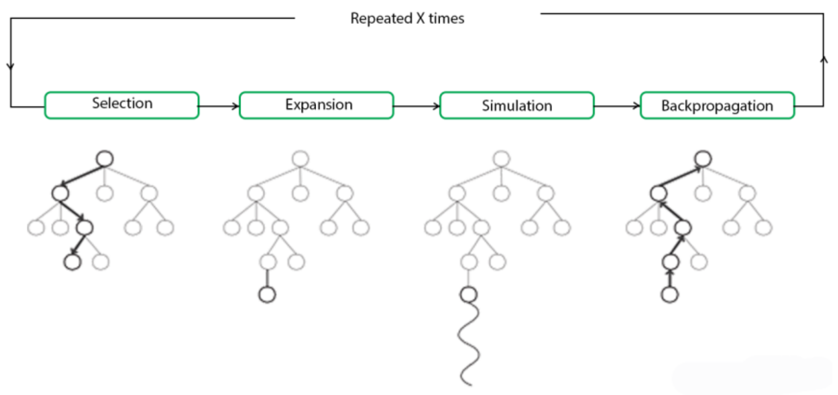
Selection: The algorithm starts at the root node of the search tree and selects a child node to explore. This is done using pure randomness (in fact not really, but for simplicity purposes we’ll consider that it’s randomness).

Expansion: Once a child node has been selected, the algorithm expands the tree by adding new child nodes to the selected node. These new nodes represent the possible moves that can be made from the current position.

Simulation: The algorithm then "plays out" a game from the current position by randomly selecting moves until the game ends. This step is called a "simulation" because it is a simplified version of the game that does not consider all of the possible moves and outcomes.

Backpropagation: After the simulation is complete, the algorithm updates the values of the nodes in the search tree based on the outcome of the game. For example, if the game ended in a win, the values of the nodes along the path from the root to the leaf node representing the winning move will be incremented.

Repeat: The algorithm repeats these steps (selection, expansion, simulation, and backpropagation) until it has explored the search tree sufficiently. The exact criteria for when to stop the search will depend on the specific implementation of MCTS.



Schema 2: each step of the MCTS by GeeksforGeeks

In reality, we made a different version of this MCTS. Indeed, we do the 3 first steps at once.

1. From the root node, we initialize a score and we randomly select moves until the game ends. At every move, the score decrements by 1 (so that less moves to win is better) and if the winner is the opponent, then the score becomes negative.

2. If the first move we made (a child of the root node) does not have a score yet, we associate the score we just got with it. If it already has a score, we add to the existing score.

3. Repeat the step 1 and 2 the number of times we want (more repeating = more simulations = more probable to find a good move if not the best).

4. Select the node that have the biggest score among the root-node’s children.

3. Benefits, limitations and statistical analysis

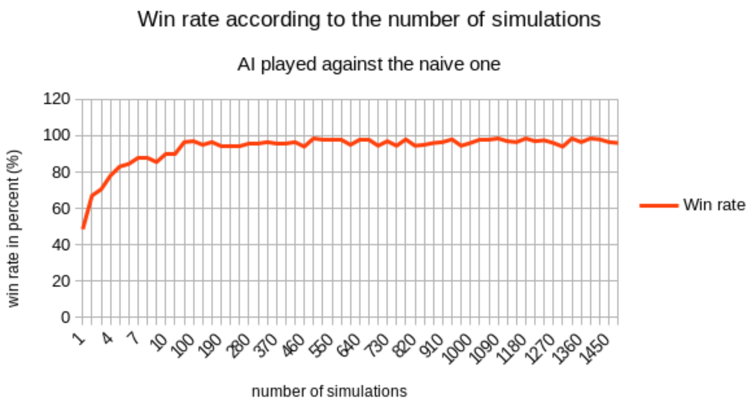
MCTS is widely used in games because it can find good moves quickly and efficiently, even in complex games with a large number of possible moves. It is also useful in other areas where it is necessary to search for the best solution among a large number of options.

The big benefit of MCTS is that you don’t have to compute a big tree containing all the states of the game. You use randomness to go on certain branches and value them. So, the more simulation, the more accurate the score is and therefore the best decision will be made.

With that simulation system, you could even for example let the player choose the difficulty of the AI (more simulation = more difficult to beat the AI, and vice-versa).

But more simulation requires also more computational power. On our personal computers, the AI started to slow nearby 10,000 simulations.

However, by studying statistically the algorithm, we showed that until a certain number of simulations, the AI's win rate is more or less the same.



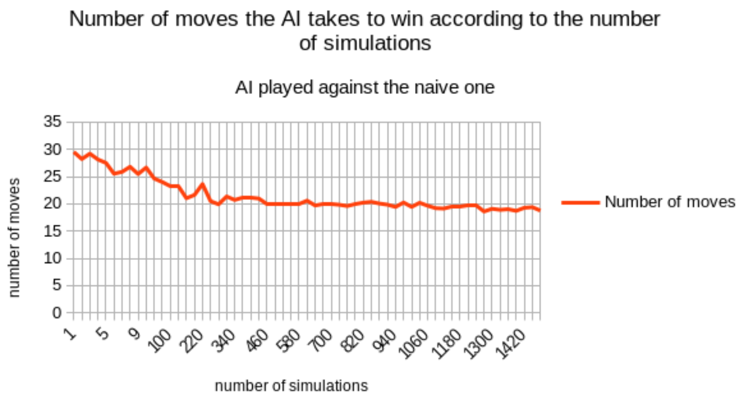
Graph 1 : For this graph (and the second bellow), we ran for each simulation 200 games in order to get a precise average. The interval for each simulation is 30 except for the 10 first that has an interval of 1.

As you see on this graph, we find a "plateau" forming from 100 simulations where the win rate is roughly the same and is almost at 100%. Meaning it's not that relevant to simulate a big number of times.

Another relevant thing to notice in this graph: when we do 1 simulation, the win rate is 50%. It makes sense because by simulating one time, we can only select one children of the root node that has been selected randomly.

Against a real player, the graph would show a win rate lower because a human being doesn't randomly move a piece but has a logic. But at least here, this graph allowed us to know that everything was working as planned.

We also made that graph:



In this one, we can also notice a "plateau" from 340 simulations. This plateau is because at same point, even if you made the best moves possible and the opponent the worst, you can't beat it in less than a certain number of moves.