Build an Adversarial Game Playing Agent

# Baseline agent

I implemented an search algorithm with incremental deepening. I used the same heuristic as the **minimax** player from the sample\_players.py file.

# Option 3: Build an agent using advanced search techniques

I chose option 3 for my evaluation and implemented the **Monte Carlo Tree Search** algorithm and compared that to the baseline agent.

The main advantage of the **Monte Carlo Tree Search** is that it does not require any heuristic function. However, that can be also a limiting factor when the opponent has access to a particularly good heuristic and can achieve good performance without searching deep in the game tree.

Another challenge about **Monte Carlo Tree Search** is choosing the depth or tree size to initiate the roll-out rather than expanding the tree. If we expand a node but due to limited time cannot evaluate all its actions many times, the estimate for the node win-rate would be similar to random roll-out. Since random roll-out does not have the overhead of expanding the tree, it is much faster, and we can have many more Monte Carlo simulations in the same time limit. So proper choice for depth can be challenging and is dependent on the processing power and time limit. On the other hand, search algorithm with incremental deepening does not require any tuning (except the heuristic) and can give the best solution considering the time-limit.

A possible solution to the above challenge is to have the choice of roll-out or expansion of an action be based on the number of visits to that node compared to its actions. For a node that is recently expanded and not all its action visited enough, we continue with roll-out. But when all the actions are visited, we start expanding the action. I **did not** implement this approach though and used a fixed depth level of 6 and minimum of 200 nodes to choose between roll-out and expansion.

I decide to not use the **fair\_matches** option, as the **Monte Carlo Tree Search** is a stochastic search and can result in a different answer every time it is used. Therefore, using the **fair\_matches** option seemed irrelevant to its performance.

The **baseline** agent was able to achieve a win-rate of **82.5%** based on 200 total games against the minimax player (100 rounds). The **Monte Carlo Tree Search** achieved a win-rate of **75%** against the minimax player for the same game conditions.

Since the **Monte Carlo Tree Search** is expected to converge to optimal solution with more time, I also ran the experiment with increased time limit to see if more time can help **Monte Carlo Tree Search** when compared to baseline. With the time limit increased to 300 ms from the default value of 150 ms, the **baseline** and the **Monte Carlo Tree Search** agents achieved win-rates of **85.5.0%** and **82%** against the minimax player, respectively. We can see that the baseline performance increased slightly, while the **Monte Carlo Tree Search** had a considerable improvement in the win-rate.