# Program explanation & Motivation of approach

**Monte Carlo Tree Search Component**

This program was an implementation of the Monte Carlo Tree search (MCTS) with modifications and additions. It worked in 4 phases:

1. Selection: A tree policy was used on nodes (representing actions leading to states) that had been seen before to select a promising path on the search tree. Upper confidence trees were used to select a promising node, where the tree was traversed downwards by selecting the current node's child with the highest UCT value, yielding a promising node at the frontier.
2. Expansion: The tree was expanded when we reached the promising node, that is, where we had no more previously seen states represented on our search tree. The promising node had 10 children generated by randomly selecting 10 legal moves and updating the board states, creating new child nodes. We did this instead of selecting all possible moves from the current board state to save on computation. The initial intent was to select these moves according to a policy rather than randomly, effectively pruning less-appealing moves from the search tree.
3. Simulation: From this newly expanded node, were randomly selected a child to play simulations, also called rollouts, using a default policy for both players. This took the form of light playouts that deployed a random strategy.
4. Backpropagation: After the simulation terminated, we update the win score and visit count attributes for states visited during selection and expansion.

**Heuristic Selection Component**

In addition, the first two moves were selected from 4 potentials, using a heuristic developed from playing the game to get domain specific knowledge and identify useful features: That there are (12 vertical +12 horizontal + 4 diagonal =) 32 zero-sum winning placements, such that a player holding one made a win in this position more likely while also robbing the opponent of the opportunity. Thus, the

first moves were selected to maximize our gains by this heuristic and hold 6 winning placements each, in the centre of each quadrant, in a way unaffected by rotations and flips, guaranteeing a permanent effect on the environment.

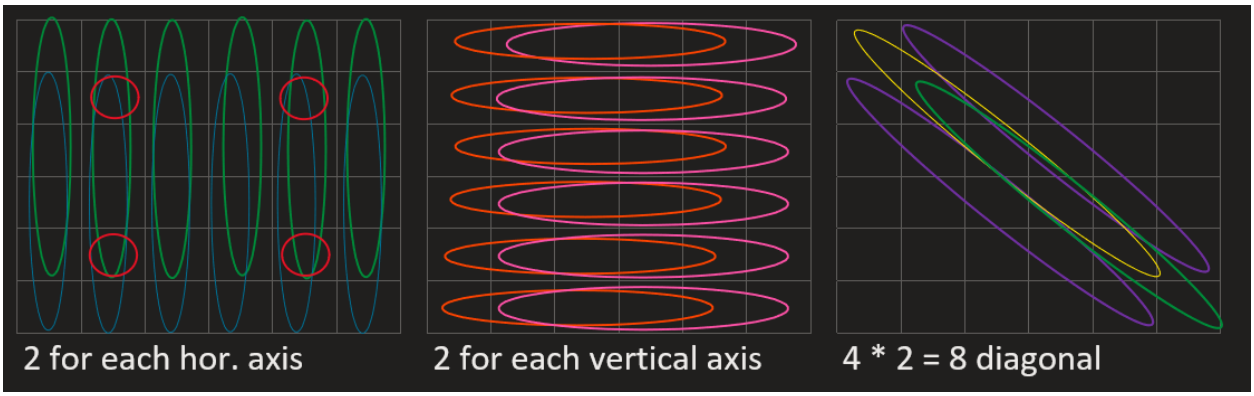


*Figure 1: An illustration of the zero-sum win-states that form the basis of this heuristic. Red circles are the 4 potential first moves.*

**Motivation**

This game has a high branching factor, where for each state we have maximum of 36 piece placements, a selection out of 4 quadrants, and 2 possible actions within each, for a total of (36)(4)(2) = d = 288. The high branching factor, alongside stochastic environment brought about by quadrant alterations, and the necessity of an evaluation function shared by the opponent for optimality (especially pitted against other students), made minimax a less appealing option.

This strategy essentially relies on utilizing our computational resources not only for performing searches in a search tree fully articulated by us, but wields randomness to assess probabilistic outcomes, and balances the tension of exploration and exploitation using a trusted Upper Confidence tree. Given our constraints of 2 seconds per move, MCTS also provided a convenient method of allocating computational capacity that could return the best estimation of move even when cut off by external constraints. Original motivations included using reinforcement style learning to better select potential moves in the expansion phase, further concentrating computational power in the simulation phase, during rollouts, to more promising moves (see *Other attempted approaches).*

## Theoretical basis of the approach

This approach used the simulation capacity of MCTS to our advantage, with a dash of heuristic guided direction. The MCTS algorithm was selected due to the high branching factor, which made articulation of potential search tree paths in the form of conformant planning impossible to naively implement. Additionally, the rotation and flipping of quadrants made for a stochastic environment that made a good evaluation function much more difficult to implement. MCTS used randomness to our advantage, here we balanced our computational resources between exploration of potential plans and exploitation focusing on seemingly good plays using Upper Confidence Trees (UCT):

$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}}$$

UCT's allowed a traversal of a the previously explored search tree in a manner that generated a value scheme for nodes that ensured a balance between exploitation and exploration. The UCT value, $Q^{\oplus}(s,a)$ was defined as the value of taking action a in state s, defining an upper bound for a confidence interval for the value of a in s, giving a bonus to actions we haven't explored much. It was composed of the $Q(s,a)$ exploitation term, utilizing information we had already gathered from rollouts for a current estimate and the exploration term, $c\sqrt{\frac{\log n(s)}{n(s,a)}}$ , giving bonuses for uncertainty and motivating exploration of unseen paths to avoid getting stuck in suboptimal strategies.

This implementation originally planned to tweak the MCTS algorithm to be more selective in its tree policy, where instead of relying totally on UCT's value schema for exploration and exploitation balance, we would expand nodes more discerningly: Alternatively, we would expand a subset of child nodes representing board states according to a state-value function gleaned from reinforcement styled learning, where the value of an action was taken as the updated sample average with regards to the reward (potential win) obtained over many plays (see *Other attempted approaches* for more details).

This would further ensure that we were spending our search efforts on more promising plays, effectively pruning sub par moves from ever making it onto the search tree and focusing capacity on more promising plays.

## Advantages & Disadvantages of approach

Advantages of this approach included avoiding reliance on an evaluation function, especially in the case that algorithms such as minimax require both players to be playing according to the same evaluation function to ensure optimality. This approach also did not necessitate a lot of domain-specific knowledge and expertise, which good evaluation functions may rely on. (Though the initial two moves did deploy a basic heuristic for selection). In addition, due to constraints articulated in terms of seconds, MCTS offered an effective method for giving the best possible estimate for a next move with an abrupt cut off point, allowing us to avoid the variability of resource usage and indeterminant termination time present in other algorithms.

Disadvantages of this approach included the approximation design choice to only expand an arbitrary subset of potential nodes, which potentially leading to subpar outcomes if the optimal move was not present in the expansion. There was also no notion of offensive and defensive balance outside of the natural trajectory exploration of the simulations: For instance, if the opponent was one move a way from winning a game, it's plausible that the defensive move to block the win could have been lost when the expansion subset was selected.

## Other attempted approaches & Future improvements

The original intent in implementing MCTS was to create a strategy that could utilize a reinforcement learning framework in the tree policy. The idea was to create an unsupervised learning capability to generate a state-value function that could ultimately aid the MCTS with a policy during the expansion phase: Selecting the highest valued actions from a particular state, rather than generating a

subset of random, legal actions (or all legal actions, which was costly with such a high branching factor). In a *learning session* over many games, each board state would be saved alongside a list of possible actions from that state. Rewards for each action would be calculated based on whether the game was ultimately won, and how many moves away such a win was, to give a value for each action. The state-value function would seek to maximize the expected utility of these cumulative rewards, and this data would be saved to disk. A policy would be born: In a *playing phase* the list of relevant board states (according to turn number) would be loaded from disk to enable the policy guided selection of the next moves generated during an expansion phase: During the expansion phase, the selected children subset would be the 10 highest valued states that were legal, with that value learned from pre-processed actions. Time constraints on this assignment, the time penalty of IO tasks and searching large lists ultimately led me to abandon this approach.

This agent could have been improved by other techniques as well: Rapid Action-Value Estimation could have improved the MCTS by sharing information across related nodes, especially in a context where quadrants were already defined as part of the game. With the assumption that it was the move that mattered rather than its locality, this could have led to more efficient information utilization in selecting promising moves. Heavy rollouts rather than light ones would also have benefitted this approach, where instead of using a random policy for move selections in simulations, heuristics were deployed to yield more relevant statistics about the relationship between a potential move and the game outcome. In addition, instead of using a time-costly file read and write system for reinforcement style learning, a Markov Decision Process implementation, making use of policy evaluation and policy improvement algorithms could have yielded a more computationally and time efficient method for creating a state value function, that could have been used on its own or in the selection of next moves in the expansion phase of the MCST.

# Sources Referenced

A Simple Alpha(Go) Zero Tutorial

https://web.stanford.edu/~surag/posts/alphazero.html

AI 101: Monte Carlo Tree Search

https://www.youtube.com/watch?v=lhFXKNyA0QA

Comparator Interface in Java with Examples

https://www.geeksforgeeks.org/comparator-interface-java/

General Game-Playing With Monte Carlo Tree Search

https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238

Java Class Structure

https://www.youtube.com/watch?v=tnJzctk-Utk

Monte Carlo Tree Search

https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Monte Carlo Tree Search for Tic-Tac-Toe Game in Java

https://www.baeldung.com/java-monte-carlo-tree-search

//Also cited in code for implementation assistance.