# COMP30024 PROJECT B REPORT

## Approach

For the first turn, we will directly select actions based on the rules of the game for the first move because the first move does not add much significance to the game, especially one with a large board. Doing so also eliminates unnecessary computation of our searching algorithm which will be discussed later.

- If the agent is red, it will select corner hex to avoid being stealed by the blue player. Corner hex is selected because it belongs to both players, selecting it means that the red player can block one of the possible destinations of the blue player. Since all four corners of the board are equally good for the initial board, the red player will select the top right corner.
- If the agent is blue, it will perform a swap if its opponent which goes first selects the centre hex. Otherwise, the blue player will select a hex in the centre of the board to create advantage.

For the second turn onwards, we applied Monte Carlo Tree Search for selecting the most optimal action. Regarding the branching factor of the game, for a 5x5 board, there are up to 25*24 = 600 possible combinations only for the first two moves and it gets exponentially larger for larger boards. Given that the branching factor is very high and evaluation functions are hard to determine, using Monte Carlo Search is beneficial (Stuart & Norvig, 2002). The algorithm basically consists of four main stages:

- Selection: From the root node, we chose a move that has the most weight which is calculated by UCB1, repeat the process down to the leaf.
- Expansion: Generate all possible children of a selected node
- Simulation: Perform playout from generated child node, moves by each player will be chosen randomly given the available actions that player can make
- Backpropagation: We use the utility which is calculated at the end of the simulation to update the tree nodes up to the root. The utility is the number of wins.

We still had to have a selection policy which in our case we used the UCB1 algorithm (Auer, Cesa-Bianchi & Fisher, 2002) as can be seen in Figure 1. This algorithm is a function that combines the utility (number of wins) with the number of times we tried for the node (a potential action for the current turn) and total number of actions taken. There is an exploration parameter (C; theoretically = sqrt(2)) to balance the exploration and exploitation. This combination in the formula ensures that the exploration (states with fewer playouts) and exploitation (states that have high win rates) is evenly balanced to get a more accurate estimated value when selecting the best child nodes (consequent actions).

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\textsc{Parent}(n))}{N(n)}}$$

Figure 1: Formula from the textbook on UCB1 (Russell, S., & Norvig, P., 2002)

## Performance evaluation

Although we used a non-traditional approach towards this assignment, we believe that our algorithm is as competitive as others. We found that Monte Carlo Tree Search is widely used in games where finding evaluation functions estimating game outcomes cannot be easily established or when the branching factor is very high. MCTS does not rely on heuristics as by random simulation it figures out the game itself and finds the best optimal move with accuracy increasing over every simulation played during the game. If simulations are infinite, the MCTS will eventually converge to a minimax tree which makes MCTS optimal (Cameron Browne et al., 2012).

As simulations can be performed infinitely until all options are exhausted, we had to make sure the time constraint was met. Given a tight time limit of $n^2$ seconds per player, per game, we had to limit our simulations to be done under a certain amount of time but not limit it too much that it affects the decision making significantly. This was done through trial and error. To make it most optimal for n=8 and n=9, we found that putting a time limit of 1.92 seconds for the simulation per turn would be ideal (We found that on average, when battling against our own algorithm, it takes about half a board size turn from each player to declare a winner. So, in addition to giving a little freedom to the time constraint, as time taken to win could differ against different players, (although this is not optimal) we put the limit for each turn to be 1.92 seconds.

As for the space constraint of 100MB, we had absolutely no problem. As the simulation stage does not store memory and just runs, we are able to use the memory at an absolute minimum. There are no issues at all with the space. After running our algorithm on many different sizes, we see that the memory space it takes is always below 1MB.

After we built our algorithm, we created a simple random algorithm (randomly picks) to test it on. Playing the majority of the games on board sizes 8x8 and 9x9, our algorithm won on average 85% of the time (25/30 for 8x8 and 26/30 for 9x9). Of course, if the time constraint was removed, with more simulations played, the win rate would drastically improve.

## **Limitations**

The efficiency of the Monte Carlo Tree Search depends largely on the number of simulations performed on a generated child node. However, the time constraint limits the number of simulations (1.92 seconds). For large boards where the branching factor is very high, the limitation is even more significant because not only the number of simulation for each node is largely reduced, but also some nodes generated later which may be good might not be considered for simulation. Therefore, we think that it is better to select good moves for nodes that are chosen for simulation to achieve better estimation. We experimented with some simulation strategy (other than randomness strategy) such as prioritising moves that:

- Results in a capture mechanism
- Have the shortest distance to the opposite sides of the same colour
- Create a connection for the nodes of the same player on the board
- Are located on the edges or corners of the board

We have also considered other strategies like incorporating heuristics together with MCTS such as thomson sampling which incorporates Bayesian concepts or Rapid Value Action Estimation evaluation function to build upon the UCB1 formula allowing the algorithm to converge faster (Maciej Świechowski et al., 2021).

However, as these strategies take up much time, the number of simulations for the chosen nodes are significantly reduced. This makes the decision of the algorithm to be restricted to those small numbers of nodes only while missing large portions of nodes that may lead to win in the end. Therefore, we stuck with the randomness strategy to allow a larger number of simulations per node.

Overall, our algorithm tries its best with the limited time constraint to pick the most optimal action every turn. We emphasise again on the time constraint as MCTS is heavily dependent on the number of simulations run. If the time constraint was removed, we believe that our algorithm would be the most optimal (in addition to enhancement to the MCTS).

## Reference Lists

Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S., 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1), pp.1-43.

Kocsis, L. and Szepesvári, C., 2006, September. Bandit based monte-carlo planning. *In European conference on machine learning* (pp. 282-293). Springer, Berlin, Heidelberg.

Świechowski, M., Godlewski, K., Sawicki, B. and Mańdziuk, J., 2021. Monte Carlo Tree Search: A Review of Recent Modifications and Applications.

Russell, S., & Norvig, P. (2002). Artificial intelligence: a modern approach.