1. Algorithm

To begin with, we developed a random move agent and greedy agent to help us better understand the classes and functions given, as well as game playing in Part B.

Then, we simultaneously worked on developing two possible adversarial game playing algorithms: Minimax and MCTS.

MCTS

We first developed standard MCTS as discussed in the lectures. However, if MCTS only expands one child node, then it can only play that child node regardless of the simulation outcomes, which renders it no different to just a random agent. (in our implementation at least)

Random expansion & playout + multi-expansion

So then, we tested expansion of 2 child nodes randomly with random playout. As expected, upon observation, most of the simulations ended due to reaching max no of turns. Almost all wins were due to luck rather than a strategic series of moves, and hence the evaluation of a node's utility (using UCB with no of visits, no of wins) from backpropagation was pointless.

Heuristic guided expansion & playout

We needed knowledge to guide MCTS to develop a strategic way to choose a move. So we incorporated an evaluative function using heuristics in expansion and playout. Using a priority queue, we picked 2 moves with the highest utility value to expand, and the move with highest utility value to play in simulation. However, while this allowed the agent to make more optimal moves, it also meant that simulations took way too long to play to a terminal state, as we had to calculate the heuristic of every possible move, to choose the best one from in expansion and in each turn for playout and due to the exponentially increasing branching factor.

Simulation Cut-off

To resolve the long runtime issue, we did research into modifications for MCTS and decided to use cutoff to limit simulation depth, which should drastically improve runtime but still provide a relatively accurate estimation of the utility of a state.

After some trial and error, we limited playout depth to: 5 turns, 3 rounds, (6 turns in reality, expansion is the first turn) and changed the evaluation for playout. To evaluate a playout, we calculated utility via our evaluative function at the beginning of playout and at the end of playout, counting a win if end utility is greater, a loss otherwise. Using 10 playouts for each turn, our MCTS agent was now able to beat the random and greedy agent.

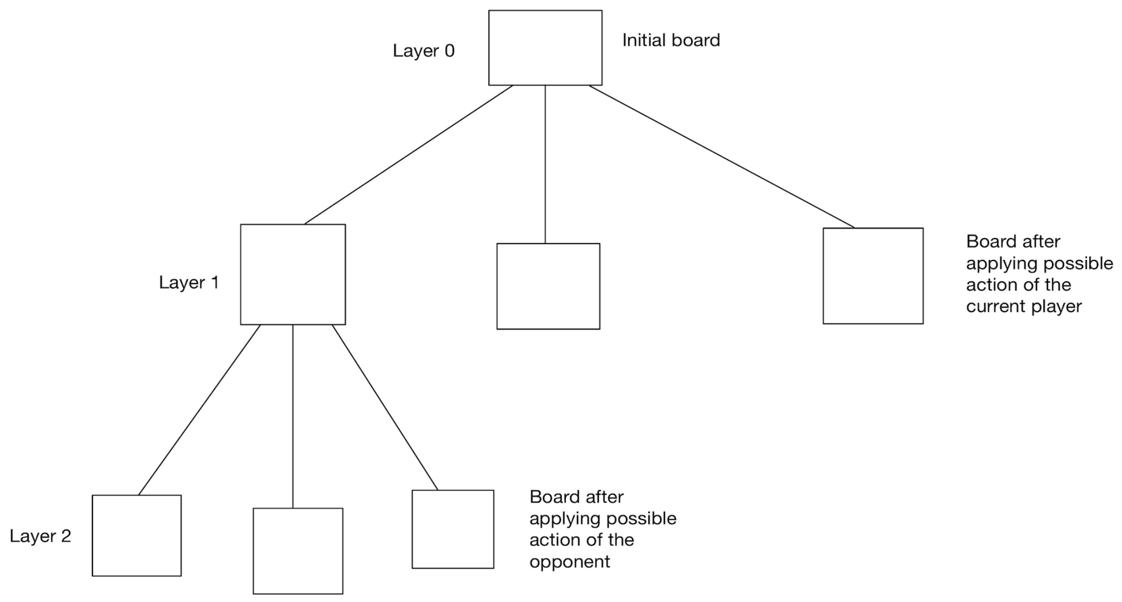
Minimax

Considering the time and memory constraints, we initially aimed to create a tree with depth of 4 to select the most optimal move, however in practice, it often exceeds the time limit, so then we reduced the depth of the tree to two layers.

The minimax will traverse through the tree, on every layer where the current player's action is applied, the algorithm will choose the node with the lowest heuristics value; on the layers where the opponent's action is applied, the algorithm will choose the node with the highest heuristics value.

Data Structure

A tree structure was used to store all of the possible actions for the current board as well as their corresponding possible actions for the opponent. Each node of the tree contains a board, the action that was recently applied to the board, a list of next possible actions, chosen action and chosen heuristic from the layer below.


Layer 0
Initial board
Layer 1
Board after applying possible action of the current player
Layer 2
Board after applying possible action of the opponent

After decreasing the depth of minimax to 2, the agent still occasionally times out in scenarios that are more complicated (many pieces on board).

Alpha-beta pruning

To resolve this issue, we introduced alpha-beta pruning. With alpha-beta pruning, child nodes which lead to a lower utility state (calculated by heuristics) than that of the parent node will be pruned. This technique significantly improved the runtime of our algorithm and it no longer timed out in our testing.

We tested our alpha-beta agent against the greedy agent we developed as well as a random agent. Alpha-beta agent is able to quickly beat the greedy agent, however it doesn't always beat the random agent. This was to be expected, as the Minimax algorithm assumes that the opponent will always take a utility maximizing move, which does not describe the behavior of a random agent, hence the weakness in our algorithm.

Beating random moves

To resolve this issue, we modified the algorithm to also consider the current best action: the possible action tree now only includes actions that have a heuristic that is less than the visited most optimal possible action in the first layer, but each node will still include all possible actions for layer 2. This way, the first layer will be selected using a 'greedy' approach, then all of the possible greedy actions will be filtered by the optimality of their future move. Since we reduced the possible_action tree size, the runtime and space complexity of the agent was also improved.

2. Performance evaluation

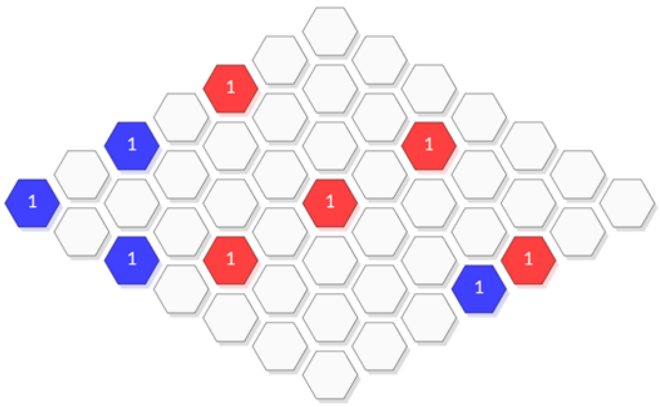
Both our MCTS and Minimax (with a-b pruning) were able to beat our random agent and our greedy agent(using different aforementioned evaluative functions).

Pitting our MCTS and Minimax algorithms against each other, while neither algorithm develops an advantage quickly, it is observed that MCTS takes longer per turn and eventually loses out to Minimax through contravening the 180 seconds time constraint. (at around 60 rounds) At that end state, it was almost always observed that Minimax had a piece and power advantage over MCTS.

Hence, we decided to go with Minimax with alpha-beta pruning as our game playing agent.

3. Evaluative function/Heuristics

Minimum number of lines the opponent pieces form



eg. 3 lines in this board

We first developed a greedy agent and experimented with different heuristics to find one that best represents board state utility. We initially continued with the approach from project a, which is calculating the minimum number of lines the opponent pieces form on the current board. Through trials of experimenting with the game, we discovered that spawning would be the preferred strategy for the initial phase of the game, thus this heuristic wasn't ideal as it is more focused on eliminating the opponent's piece and therefore may lead to a disadvantage build up in the early stage of the game.

$$\frac{Number\ of\ opponent's\ piece}{Number\ of\ current\ player's\ piece}$$

We then developed a new heuristics which compares the ratio between red and blue pieces, this new heuristic will prioritize spawning in the initial stage. Nevertheless, it does not take into account the power of each piece, therefore is not ideal when evaluating the utility of spread actions.

$$\frac{K_{opponent}^{\ N_{opponent}}}{K_{current}^{\ N_{current}}}$$

$K_{opponent} = Total\ power\ of\ opponent's\ pieces$

$K_{current} = Total\ power\ of\ current\ player's\ pieces$

$N_{opponent} = Number\ of\ opponent's\ piece$

$N_{current} = Number\ of\ current\ player's\ piece$

We further improved the heuristic by adding the power of each player's pieces and the final heuristic is shown above.

This heuristics takes both the power and number of pieces into account and is more effective for determining both spreading and spawning strategies.

Strategic enhancement

When spawning, we removed the 6 squares around all opponent pieces as possible spawn locations, as a spawn action next to an existing opponent piece early on will likely give the opponent an advantage through spreading to our piece.

After testing our Minimax - AB against itself with a different heuristic and against MCTS. We observed that for the first ~20 turns, both agents only chose spawn actions. Leading us to conclude that the ideal early game strategy is to spawn unless a spread is able to win the game.

Hence, to improve runtime, we set our agent to only consider spawn actions for the first 20 turns (10 per agent), this way Minimax does not have to build a tree for the first 10 moves. Drastically improving runtime without sacrificing much strategic performance.

References

Finnsson, H. (2021). Generalized Monte-Carlo Tree Search Extensions for General Game Playing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1), pp.1550–1556. doi:https://doi.org/10.1609/aaai.v26i1.8330.

Winands, M.H.M., Bjornsson, Y. and Saito, J.-T. (2010). Monte Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4), pp.239–250. doi:https://doi.org/10.1109/tciaig.2010.2061050.