1. Introduction

In the given assignment, we were tasked with developing an Artificial Intelligence program revolving around the game *Infexion*. The goal in the AI is to be able to perform a form of adversarial search, in order to win a game through a sequence of *Spawn* and *Spread* actions. To tackle this project, our group decided to use the Monte Carlo Tree Search (MCTS) algorithm.

2. Approach to MCTS

2.1 Reasoning

Our group chose Monte Carlo Tree Search as a way to tackle Infexion, as it is a game that can be described as one of *perfect information* and characteristically *deterministic*, making it a viable option for MCTS. The choice of MCTS is further supported by the fact that Infexion is a novel game, hence the group members of this project are not well-experienced with the ins and the outs of the game, which would make designing Heuristics and Evaluations difficult. MCTS helps us with this as fundamentally, the algorithm can work purely on playing random simulations of every state of the game and return a good solution.

2.2 Components of MCTS

The algorithm can be divided into 4 main components, namely: *Selection*, *Expansion*, *Simulation*, and *Back-Propagation*.

Selection uses a strategy to choose and generate Child Nodes (states) until hitting a Leaf Node. The strategy used in our program was to continuously update a *searchNode*, and select it once it has been determined to be fully expanded and not marked as having a winning state.

Expansion in basic MCTS expands a node's state into all its possible moves/actions. Given the constraints in this assignment, we have decided to include some heuristics and conditions to reduce the nodes we expand, making the algorithm more cost-efficient. In our case for example, we expanded nodes that immediately led to a more advantageous state.

- Only allows for spawn/spread actions that aren't able to be taken by opponent in a subsequent move, unless it has an adjacent piece that can defend it
- Prefers spread actions that increase the overall power of the player

Simulation performs a random playout of every possible action's resulting *board-state* to a *terminal state* under normal circumstances. We achieved this by going through a collection of *legal moves*, and simulating a game through those. Under our implementation, we reduced the randomness, making it so that the *getLegalMoves()* function only considers moves that are beneficial to the current player.

Back-Propagation

Back-Propagation makes use of the outcome received from the *Simulation* phase. Usually, this is done by incrementing the number of visits and wins according to the result. In our case, we did this through the Node's *update()* function to increment the number of visits and add a cumulative score according to the difference of *power* between the two players from the simulation. The use of a cumulative score was used with the goal of encouraging faster winning-solutions with a higher margin of victory.

3. Implementation Performance

As a solver, our implementation of the algorithm runs effectively to return a winning-solution. We have only been able to test it against an Agent that randomly Spawns and Spreads, which has resulted in a 100% win rate so far.

Regarding effectiveness in comparison to other search-algorithms, theoretically, MCTS should be better-suited to this form of problem than the likes of A* Search, as being an

Adversarial Search algorithm allows it to make moves in response to an opponent's actions. A* Search and other search algorithms like it would only be suited in the case of a non-changing board.

Compared to another Adversarial Search algorithm, *MiniMax*, we believe MCTS should perform better as MiniMax only returns optimal solutions assuming the opponent is also optimal. With regards to *Alpha-Beta Pruning*, we also believe that theoretically, MCTS should fare better as there is no need for a cut-off depth. The game of *Infexion* could be argued to need this depth due to the large complexity of the game, in terms of how many moves can be made and its resulting states.

According to Jin, Y. & Benjamin, Shaun (2015), the Time and Space Complexity can be computed by:

$$Time\ Complexity\ =\ O(\frac{mkI}{C})$$

$$Space\ Complexity\ =\ O(mk)$$

m: $number\ of\ children,\ k$: $number\ of\ child's\ simulations,\ I$: $number\ of\ Iterations,\ C$: $cores$

With regards to our implementation, while time and space complexities should be the same as a regular MCTS, ours should be faster due to the aforementioned heuristics and optimisations implemented in *Section 2.2*.

4. Supporting Classes

4.1. Node Class

To facilitate Monte Carlo Tree Search, we created a *Node* class within *mcts.py*. It contains the *state* of a board, *parent*, *children*, number of *visits*, and *score*. Many implementations of MCTS use the number of *wins* over a cumulative score, but we decided to use a score to encourage selecting states with higher margins of victory.

4.2. GameBoard Class

We created a GameBoard class to hold various states of the board and relevant information. Functions defined within this class are primarily used to process or evaluate the board for MCTS' processes, such as *updateBoard()*, *getWinner()*, and *getLegalMoves()*.

Bibliography

Jin, Y., & Benjamin, S. (2015). *Monte Carlo Search Tree Report*. https://stanford.edu/~rezab/classes/cme323/S15/projects/montecarlo_search_tree_report.pdf