Development Report

Game Development - Gade7321

Francesca Fitzgerald ST10143178

2024

## Chosen Behaviour

**Chosen Behaviour & Motivation:**

I decided to use a Monte Carlo Search Tree (MCST) algorithm as my ad-hoc behaviour in this game. Because my game does not have finite game states or game moves, using a MCST algorithm is far more effective and efficient. Using a minimax algorithm would have required me to calculate and record every possible game state in a data structure, requiring a lot of memory. I chose to use the MCST method because it can simulate the game and access the necessary information to make a decision, each time the AI needs to move or evaluate the game state.

The game can go in indefinitely, much like *Chess*, because of how the game is structured. This is the main reason I decided to use the Monte Carlo method. I have done research on what games use MCST verses Minimax, and an MCST algorithm is often chosen due to the level of strategy required in chess-like games (Medium, 2024).

**Implementation:**

In part 2, I derived a utility function to be used to influence my AI’s decisions. However, because I used an ad-hoc behaviour algorithm (not a machine learning algorithm), I did not need to implement this utility function. This was confusing as the brief specifically states that we are using ad-hoc behaviours in this POE, yet it asks for components needed in machine learning. Ad-hoc behaviour never needs reinforcement algorithms (like a utility function).

The selection phase of the MCST algorithm requires the implementation of the Upper Confidence Bound (UCB 1) formula:

The first term evaluates exploitation, and the second term evaluates exploration. The code in my program uses two parameters to determine which game state or node to select: the difference in wins verses losses from that game state, and how many times that node has been visited. This formula dictates which nodes the algorithm will choose to explore. The above formula is used in step 1 of the MCST algorithm – the selection phase.

The next phase (step 2) is expansion. Here the algorithm takes the chosen node and expands all the possible child nodes until it reaches the terminal node, where a game result achieved.

In step 3 (simulation phase), the game is simulated as if it were being played out, with the moves recorded for each game state. The resulting terminal state from the chosen path of nodes is recorded as either a 1 (a win), -1 (a loss), or 0 (a draw).

From here step 4 (backpropagation) is carried out. The program performs a backpropagation function that retraces the nodes that lead to that terminal state and updates each node with necessary data.

From here, the AI can decide what moves will result in the desired game state – usually a win.

Coding an AI algorithm in C# had its difficulties, especially since most resources online are aimed at Python being used for these types of programs. It required even more of an understanding of what my program needed to do, as I had to code in my own methods to replace the many Python libraries. This did lead to me learning exactly how this algorithm needed to work and how it needed to communicate with my pre-existing game.