Comp30024 assignment 2

Duc Huy Le - 1254871

1 Introduction

We have developed an agent capable of playing the game Infexion with the aim to win. The agent operates based on the state of the game, from there, with the Minimax algorithm, it will find its possible moves, analyse how opponent will react to each move, and find the optimal course of action while being restraint to 180 seconds and 250MB memory usage per player.

2 Strategies

We use a simple strategy that our agent try to capture opponent node one by one.

2.1 First move

For the first move, our agent will place a spawn in the middle of the board, since the board is infinitely repeated, any point on the board can be considered the middle. Our strategy focus on counter play the opponent instead of trying to eliminate opponent in the shortest way.

2.2 Subsequent moves

For every subsequent move, we have 3 things to do every turn:

1. Generate where it can spawn and can it capture opponent pieces. Calculate how the opponent would react to those move, assuming the opponent is playing optimally.

2. Evaluate those move.

3. Using Minimax to find the optimal move.

For task 1, we use the Board() dataclass to store the current state of the game. Also, it stores the state of the game after each possible moves by manipulate the apply_action() and undo_action() function to the Board().

For SpawnAction(), our agent is built to only check the neighbors of hexes that are already on the board. For every opponent's node.It help speed up the process by reduce the number of nodes need to be evaluate, even though it has a risk of misisng out the optimal move.

For SpreadAction(), out agent will spread to the unprotected node of opponent on the board. By doing this, we secure that everytime the agent capture a opponent node, we do not lose any thing.

To evaluate each state of the game, we simply use the difference between the power of players:

`Eval() = (currentRedPower - currentBluePower) * turn`

Note: turn has value of -1 or 1 depend on the current turn color of the board

We ensure that we consider the state of the game for each players' perspective by multiply the Eval() to turn. Otherwise, the agent might choose a move that is not optimal and loss the game. As the game continue, our agent will increase the number of look ahead moves based on the total power of the current board. At any given state of the game, there will be subsequent moves that lead to the same result, checking the states of game from those moves is unnecessary and computationally expensive. Alpha Beta pruning can help reduce branching factor significanlly with good move ordering, leading to higher depth in the same amount of time. Our implementation of the function MINIMAX, with Alpha Beta pruning

```
function MINIMAX(depth, alpha, beta, turn)
    if $depth == 0$ then return $Eval()$
    end if
    $moves \leftarrow generateMove()$
    if $turn$ then
        $bestEval \leftarrow -\infty$
        for $move\ in\ moves$ do
            $self.board.apply\_action()$
            $eval \leftarrow$ MINIMAX($depth - 1,\ alpha,\ beta,\ False$)
            $bestEval \leftarrow max(bestEval, eval)$
            $alpha \leftarrow max(alpha, bestEval)$
            $self.board.undo\_action()$
            if $beta\ \leq\ alpha$ then
                break
            end if
        end for
        return $bestEval$
    else
        $bestEval \leftarrow \infty$
        for $move\ in\ moves$ do
            $self.board.apply\_action()$
            $eval \leftarrow$ MINIMAX($depth - 1,\ alpha,\ beta,\ True$)
            $bestEval \leftarrow min(bestEval, eval)$
            $beta \leftarrow min(beta, bestEval)$
            $self.board.undo\_action()$
            if $beta\ \leq\ alpha$ then
                break
            end if
        end for
        return $bestEval$
    end if
end function
```

3 Performance evaluation

3.1 Time and space complexity

3.1.1 Spawn action

A protected node is a node that is placed within other nodes protected zone. A protected zone is the zone that our agent can capture on next turn. The function is used to calculate the protected zone of a player is protected_zone(). By iterating through every node on the board and every possible direction of the game. We store the zone using defaultdict, from that we can reduce the access cost of it to $O(1)$. The cost of calculating protected zone is $O(3i)$ where i is the turn.

Our agent only do 2 type of spawn action, which is spawn outside of opponent reach range (or protected zone) with $O(3i)$, and a protected cell in opponent reach range $O(3i)$ (i is the turn). The total cost of calculating spawn action for each turn is $O(i)$. To spawn outside of opponent protected zone, our agent iterate through every opponent's node and use the same technique as calculate protected zone but increase it coverage by 1. The extra layer is where the node get to spawn. On the other hand, for every turn, our agent check if there is node outside opponent's protected zone, if yes, it will spawn a node inside the joint of its protected zone and opponent's protected zone.

Our agent return only 1 action for each time it calls the spawn_node() function. By doing this, it reduces the branching factor to 1 and increase the speed of calculating minimax. However, it creates a trade off of accuracy for the speed.

Experiment: this is not included in our submission, however, we want to include this in our report. In order to increase accuracy, instead of just return 1 spawn action (agent A), we calculate all the possible joint between 2 players $O(2 * i)$ (i is the turn) (agent B). Agent B can beat up agent A with the same depth (for depth = 5, B is dominating A), however, A won when depth of A = 9 and B = 5 and 2 agents consume the similar amount of time to calculate for next move.

3.1.2 Spread action

As for spread action, the agent calculate all possible capture on current board, and store it to a list. Let i be the number of turn pass on the board, there are at most $\lceil \frac{i}{2} \rceil$capture moves.

3.1.3 Board variations

Instead of using a tree to store all the possible state of the board, we take advantage of Board() class. We use the Board() class to store the current state of the game, and a list to store all the possible moves on that state. Let i be the turn and d be the depth of minimax, there can be at most i occupied hexes on the board, so most the number of hexes we need to check is 6i. In reality, this number can be significantly lower because node will be captured, or spread action that increase power of a cell from 2 clustered node. At each state of the game there are at most $t_i = (2 + \frac{i}{2})^d$ move generated. Therefore the total node that the tree depth d generated after n^{th} turns is

$$t_n = \sum_{i=0}^{n} (1 + \frac{i}{2})^d$$

$$t_n < O(2^n)$$

Complexity of joining 2 lists (spawn list and capture list) into 1 is $O((2 + \frac{i}{2}) * d * n)$

3.1.4 Evaluation function

We implement a simple evaluation function for our agent by just calculate the difference of power between 2 players. Because of using this simple evaluation, our agent has some limitation when implement Minimax algorithm. First of all, the depth we use for Minimax has to be odd. This allow the agent to calculate how the opponent react to specific moves it generated. Time complexity of this evaluation function is $O(i)$ (i is the turn). Moreover, by reduce the number of moves generated by each turn, we sacrifice the accuracy of optimal move for the speed of computing minimax.

3.1.5 Minimax with cutoff and Alpha-Beta pruning

Let n be the number of nodes currently on the board, without Alpha-Beta pruning, running time complexity would be $O(n)$ because the Minimax algorithm will recursively evaluate every move generated. On the other hand, with Alpha-Beta pruning and good move ordering, in our case, capture moves are placed before any spawn moves, it can reduce down to only $O(n^{\frac{1}{2}})$.

3.2 Game performance

As the game begin, there are plenty of time and the amount of nodes on the board is relatively small, this allow agent to have a higher number of look ahead moves. For number of look a head moves of 9, our agent can yield better result than a greedy or random player most of the time.

With number of look ahead moves from 7 to 3, the agent can still make reasonable decision as it can see how the opponent can react to its move, and decide which course of action to take.

With number of look ahead moves of 1, the agent will lose significant performance accuracy. This is due to our evaluation function, comparing the different in power between 2 players is not enough to evaluate the opponent reaction to our moves. If the agent blindly trying to capture opponent node without considering how this node is protected, the opponent can take away its node immediately.

Overall, the agent can perform relatively well when the number of look ahead moves more than 1. Other than that it is no different from greedy agent.

4 Limitation

By using a naive approach on evaluation function, our agent can only see what move is better than other moves. Which have a critical disadvantage, it cannot see how the opponent plan to conquer all of its nodes. Also, by limiting the number of nodes generated each turn, there are chances that optimal nodes are not generated, and all possible move in the list can lead to a loss. Moreover, the choice of not using a tree to store all the possible moves prevent us from using Monte Carlo algorithm. Each turn, our agent have to calculate everything all over again.

5 Conclusion

In conclusion, Minimax with Alpha-Beta pruning is incredibly useful in perfect-information game such as Infexion. However, it consumes incredibly high in time and space complexity as a trade off. Therefore, a significant optimization for both time and space are required if one would like to preserve the performance of agent to play a game with such high branching factor.