Comp30024 assignment 2

Tuan Anh Chau - 1166394
Truong Giang Hoang - 1166323

1 Introduction

We have developed an agent capable of playing the game Cachex with the aim to win. The agent operates based on the current state of the game, from there, with the Minimax principle, it will find its possible moves, analyse how the opponent will react to each of those moves, then find the optimal course of action, all while being restraint to n^2 seconds in total time and 100Mb in total memory usage.

2 Strategies

2.1 First move

For red, the first moves can be anywhere except for the acute corners, namely (0, 0) and (n-1, n-1), since they have the lowest branching factor of 2.

For blue, if the board size if less than 5, steal the move right away, because for small boards, having the first move advantage is more important than having a move with higher branching factor. For all other board sizes, steal red's move only if red does not place their first move in the acute corners. Blue purposely ignore the middle hex to steal red's hex as blue is now making the first move, and therefore can dictate the flow of the match. It is important who controls the flow of the game, if blue plays correctly, red's only choice is to block blue's move and hence unable to develop his own moves.

2.2 Subsequent moves

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

1. Map out what can you play and how the opponent would react to that move, repeated for the number of lookahead moves you want. Assuming the player is playing optimally.
2. Evaluate those moves.
3. Use Minimax to find the optimal move.

For the number 1 task, we have used a n-ary Tree to store the state of the game after each possible moves. We have modified the agent so that it only check for the neighbours of hexes that are already on the board. Although this bear the risk of missing out on the actual optimal move, it can help speed up the process by reducing the number of states needed to evaluate. Additionally, consider the below board:

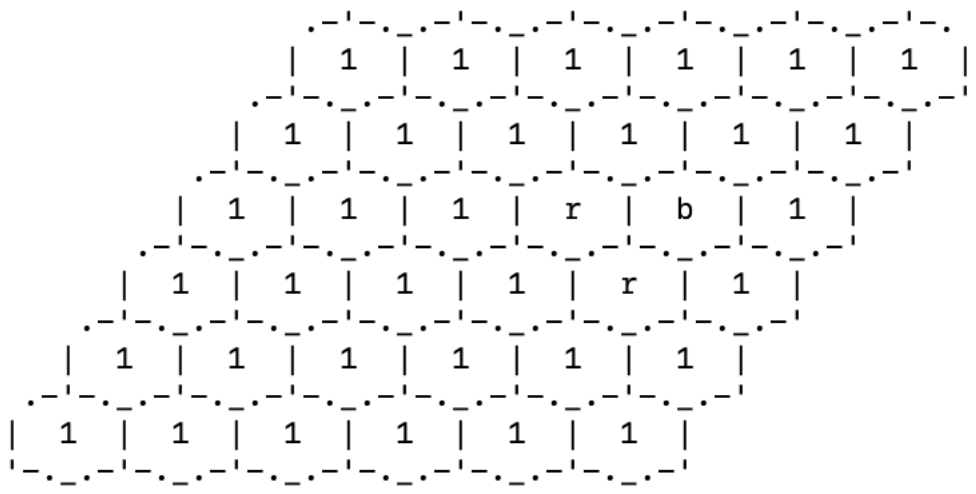

Figure 1. Sample board, the 1s are the evaluation of the hexes (Numbers are not accurate, only for representation purposes)

In this situation, every move have the same evaluation of 1, if we were to check every open hexes iteratively, we would have picked hex (0, 0), instead of playing around moves that we have already placed down to develop them.

Each node in the tree would contain these attributes:

- board - the state of the game at that node
- move - the last move the player made prior to said state
- color - the color of the player
- value - evaluation value of the state
- children - a list of its children nodes

To construct the complete state Tree, first get all the unoccupied neighbours of the occupied hexes on the board. For each neighbour, create a new Node contains a board identical to its parent, place the neighbour on that board, and assign the Node to the children of the parent Node. Repeat the process until the cutoff depth is reached.

To evaluate a state of the game, we used the function below:

`Eval(state) = numPlaysToWin(opponent, state) - numPlaysToWin(agent, state)`[1]

This evaluation function ensures that we consider the state of the game from both players' perspective, if either one is left out, the agent may be blindsighted and choose a sub-optimal move that loses the game for them.

We have implemented a BFS function to help calculate the minimum number of nodes needed to form a consecutive path between a player's 2 borders, which includes hexes that are already on the board. Despite BFS having an expensive time complexity, we chose it for the task of finding the minimum number of moves needed to win as it guaranteed the shortest path. Due to the computation limitations and the nature of BFS, we have to reduce the number of lookahead moves for large-sized board, such as from 10 to 15, this will result in a smaller state tree and less board variations to evaluate.

Our allocation of lookahead moves based on the size of the board are as follow:

- n = 3 to 5: lookahead = 3
- n = 6 to 9: lookahead = 2
- n = 10 to 15: lookahead = 1, this is the same as a greedy agent

At any given state of the game, there can be sequence of moves that lead to the same result, of moves that have similar Evaluation points, checking the states of the games from those moves is redundant and computationally expensive. Alpha-Beta pruning can help reduce the branching factor significantly with good moves ordering, leading to larger tree depth in the same amount of time. For those sequences with similar evaluation, we should prioritize those that leads to those result sooner, i.e sequences with shorter length. This is where the depth parameter in the MINIMAX function comes in, for similar evaluation metrics states, the deeper it is in the state tree, the lower evaluation it will receive.

Our implementation of the function MINIMAX, with Alpha-Beta pruning:

```
function MINIMAX(tree, alpha, beta, depth=0, is_max=True)
    if not node.children then
        if not node.value then
            value ← Eval(node.state)
            return value
        end if
        return node.value
    end if

    if is_max then
        best_value ← −∞
        for child in node.children do
            value ← MINIMAX(child, alpha, beta, depth + 1, False)
            best_value ← max(best_value, value)
            alpha ← max(alpha, best_value)
            if beta ≤ alpha then
                break
            end if
        end for
        return best_value
    else
        best_value ← ∞
        for child in node.children do
            value ← MINIMAX(child, alpha, beta, depth + 1, True)
            best_value ← min(best_value, value)
            beta ← min(alpha, best_value)
            if beta ≤ alpha then
                break
            end if
        end for
        return best_value
    end if
```

3 Performance evaluation

3.1 Time and space complexity analysis

3.1.1 Board variations tree construction

As board size n increases, the number of branch each node have does not actually increases exponentially. This is because we do not check every possible empty move on the board, as stated above, we only check the neighbours of the occupied hexes on the board.

Let i be the turn, there can be at most i occupied hexes on the board, so the most number of hexes we need to check is 6i. In reality this number can be significantly lower if the majority of the occupied hexes are on the border, or clustered together, which reduce the number of unoccupied neighbours there are. Assuming the branching factor is 3 for average case instead of 6, due to overlapping neighbours, border hexes, etc. The board currently have x occupied hexes. Capturing is ignored. At layer 0, there is only 1 node: the root node, at layer 1, there is $3x$ nodes, at layer 2, each board will have $x+1$ nodes, as the number of node on the board increase by 1 after each layer, the total number of nodes will then be $3x(3(x+1))$. Follow the rule, we can represent the number of node on layer d of the tree, t_d, with a recursive formula:

$$t_0 = 1,\ t_1 = 3x,\ t_d = t_{d-1}(3(x+d))$$

The total number of nodes of a tree depth d is simply then: $s = \sum_{i=0}^{d} t_i$

Complexity of the tree construction:

$$\begin{aligned}
t_n &= t_{n-1}3(n+x) \\
&= t_{n-2}3(n+x-1)3(n+x) \\
&= t_{n-3}3(n+x-2)3(n+x-1)3(n+x) \\
&\vdots \\
&= t_{n-k} + 3^k \frac{(n+x)!}{(n+x-k+1)!} \\
&\vdots \\
&Let\ k = n-1, \\
&= 3x + 3^n + \frac{(n+x)!}{(x+2)!} \\
&\in O(n!)\ since\ a^n\ << \ n!\ and\ x < n
\end{aligned}$$

As the tree grows at a significant pace, cutoff test for bigger board has to be low in order to maintain the space constraint as well as the time constraint, as MINIMAX is also sensitive to the size of the tree.

Further optimization can be made here. Instead of creating a new tree every move like we did, one can create only one instance of the tree and only expand the bottom layer after a move. However, since we only check for limited subset of unoccupied hexes, if the opponent choose a move that we did not anticipated, the whole function will collapse, or an entire new tree will have to be created nevertheless. Additionally, the majority of nodes are resided in the few bottom layers, so indeed the runtime is reduced, but not drastically.

3.1.2 Evaluation function

This is the bottleneck of the entire program. There is no getting around using BFS for finding the smallest set of moves to win the game. Iterative Deepening Search (IDS) do have a better runtime

complexity that would significantly increase the agent's performance, but sacrificing its accuracy, as IDS can give overestimated result of the game evaluation. Although IDS does find the shortest length of path, it have no way of checking for preoccupied hexes, as the underlying search function is Depth First Search, which does not provide an optimal path. Without the optimal path, there is no way to count the actual number of minimum moves needed to win.

Time complexity of BFS is $O(b^n)$ with b being the branching factor and n being the size of the board. The function is then called repeatedly for t_d times, with t_d being the number of nodes in the lowest layer of the tree, as we only need to evaluate those leaf nodes. So the total time complexity for only evaluating those states is $O(t_d b^n)$.

Although storing evaluated states of the board in a Hashmap for lookup can increase total runtime tremendously, it is a sacrifice of memory space for runtime. A board of size n can have at most 3^{n^2} (without consider the rules of the game or whether anyone has won yet, this result is taken from the fact that a hex can have 3 states: unoccupied, red, or blue, and there are n^2 hexes on a board size n). The amount of space required will rise exponentially with respect to the size of the board, and therefore not ideal for a space-constraint problem such as this one.

The only way to compensate for this is the reduction in the number of time the function is called, or reduce the number of node in the state trees, which is why only neighbour of occupied hexes are considered, and the cutoff is decreased as board size increased. Further more, when a board state is being evaluated, the function will loop over all the occupied hexes of the same color, perform BFS on each of them, then take the minimum value, to avoid overestimation. This can be avoided for hexes of the same color on a continuous path, because all hexes on that path will have output the same minimum number of moves to win. To achieve this, we have modified our BFS function to also keep track of all hexes that form said continuous path from the start, and assign all those hexes the same evaluation value.

3.1.3 Minimax with cutoff and Alpha-Beta pruning

Supposedly we have recorded all values of all nodes within the tree, then the actual computing time of MINIMAX is the smallest of the three. Without Alpha-Beta pruning, runtime complexity would be $O(n)$ with n being the number of nodes in the state tree. With Alpha-Beta and good move ordering, it can reduce down to only $O(n^{\frac{1}{2}})$. This also helps reduce the number of times we need to call the evaluation function BFS. Reduction in evaluation time can allow for deeper tree construction, i.e increase in number of lookahead moves, which helps in the agent's performance.

3.2 Game performance

For small-sized board with n ranging from 3 to 7, the cutoff limit for tree construction is large and therefore can yield better result than a greedy or random player most of the time. Ideally cutoff should be greater than or equal to 3, then your agent can foresee at least 2 of your moves.

With cutoff depth = 2, the agent can still make reasonable decision as you can see how the opponent would react to your move, and decide which course of action to take based on those reaction.

With cutoff depth = 1, the agent will lose significant performance accuracy. This is due to Cachex's capturing mechanism. If an agent blindly trying to build a continuous path without considering the opponent's hexes adjacent to said path, the opponent can perform capture and take away 2 of your hexes immediately.

Overall, the agent can perform relatively well when the board size is less than 10. Over that it is no different than a greedy agent.

4 Conclusion

In conclusion, Minimax with alpha-beta can proved to be incredibly useful in perfect-information game such as Hex, but is incredibly high in time and space complexity. For a game with such a high branching factor such as Hex, significant optimization of both time and space are needed if one would like to preserve the performance of the agent.

5 References

[1] Horrell, G., Echt-Wilson, E., Malik, A. (n.d.) *An AI Agent for Playing Hex.* Stanford University. https://tinyurl.com/yas7jk8h