**COMP30024 Artificial Intelligence**
**Project Part B: Playing the Game**

Authors: Ying Shan Saw (1118861) & Joeann Chong (1152260)

*Date Completed: 11/05/2022*

## Approach:

The first step to understanding how to write an AI for the game was necessary. During our course, we learned a few strategies, including pathfinding algorithms such as A* search, Dijkstra's and so on. We decided on a minimax algorithm, which seems to be our best bet since there is an "opponent" we need to predict.

The program implements minimax algorithm with a depth of 2 alongside with alpha-beta pruning that tends to minimize the time and space complexity. The minimax algorithm will look two moves ahead of the current move based on its evaluation function, and it will return the coordinates for the next best piece to be placed (based on the current board state). If a stealing move is chosen applied on player 2 with the stolen piece coordinates.

```
if depth <- 0 or game_over
        return chosen_coord, (value, steal)
if max_player
        value <- -infinity
        for coord in neighbours
                result <- minimax(depth-1, False)
                score <- result[value]
                if score > value
                        value <- score
                        chosen_coord = coord
                alpha <- max(alpha, value)
                end if
                if alpha >= beta
                        break
                end if
        end for
        return chosen_coord, (value, steal)
else
        value <- infinity
        for coord in neighbours
                result <- minimax(depth-1, True)
                score <- result[value]
                if score < value
                        value <- score
                        chosen_coord = coord
                beta <- min(beta, value)
                end if
                if alpha >= beta
                        break
                end if
        end for
        return chosen_coord, (value, steal)
end if
```

*Figure 1. Minimax pseudocode*

***Evaluation Function***

Measuring the connectivity of player nodes in Hex is crucial for quality analysis. An artificial agent will need some way of assessing how well-connected the opposite sides of the board are. This is done using the evaluation function.

$$eval(s,p) = board\_state(s,p) - board\_state(s,p)$$

*s – arbitrary position*
*p - player*

Our evaluation function incorporates three aspects:

- If the total number of pieces on board is strictly more than $\frac{n^2}{2}$ (where n is the size of board), it checks whether there are **available captures** on the board (score increases exponentially based on the number of captures)
- the **length of the connected coordinates** *(in terms of height (`maximizingPlayer`) or width (`minimisingPlayer`)*
- the **distance of the current coordinate to the closest goal area** (if it exists)

***Greedy Algorithm***

If the board size is small (n<7), the algorithm prioritizes on placing the starting pieces on either of the corners (0,0) or (n-1, n-1), and implements a greedy algorithm with direct adjacent nodes. It also considers placing a piece to block the opponent's moves (in case the opponent is the one to make the first move). This saves on time and space complexity although it is not the best algorithm.

**Data Structures & Implementation**

2 classes are introduced: Player and Board

| **Module** | **Description** |
|---|---|
| Referee | Referee validates the action if it is valid and ends the turn by informing the player of its action played. |
| **Class** | |
| Player | Player decides what action to perform for the current round based on the minimax algorithm |
| Board | Sets up the board by storing related information: board size, board state |

**Method:**

1. For both maximizing and minimizing players, the minimax function is recursively being called to find the next best piece to place on the board based on our evaluation function. We further implemented alpha-beta algorithm on minimax for improvisation, it speeds up by pruning the tree.
   An example figure of Cachex game using minimax alpha-beta pruning with depth of 3 is as follows

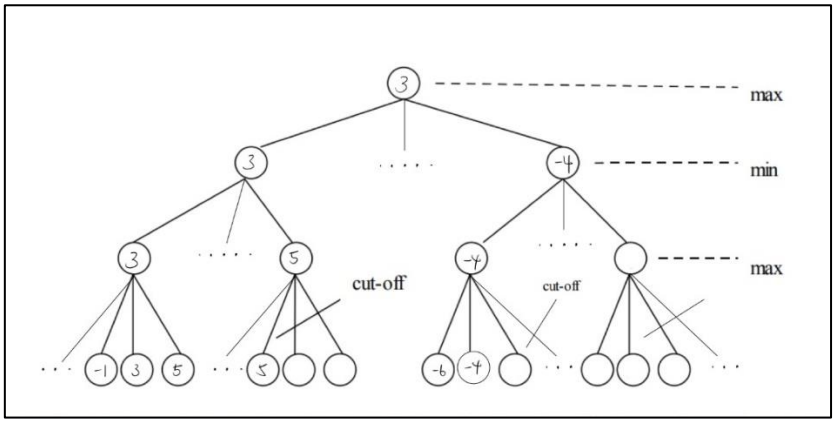



*Figure 2. alpha-beta minimax presentation*

To determine the next best node for the maximizing player (root node), the values assigned to the leaf nodes that are calculated using an evaluation function must be passed back through the tree from the leaves to the root. (Galli, 2019)

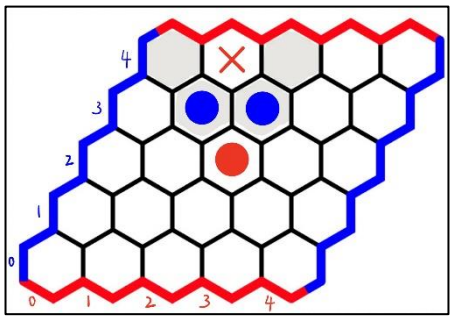
2. The main rule for Cachex game is to reach the goal from the other side with an unbroken chain. Therefore, it is important to construct an evaluation function for the connectivity measure.
   Suppose the base evaluative function for a Cachex game without capture and steal is the following:

   *f(max) = # of captures + # of connected pieces + distance goal to either side*

   *where max is the maximizing player*

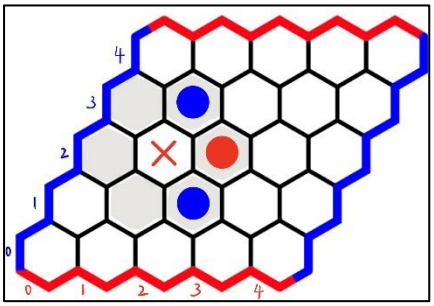
   similar approaches for f(min), providing a few tactical advantages. One of which it's calculated based on the number of each player's pieces on the board since the size of board may differ in each round. Hence, *f(min)* and *f(max)* gives an estimate of the value of the next best piece that is suitable for maximizing and minimizing player to place on the board respectively. We believe that this evaluation function is a reasonable strategy for a fair game.

3. For the capture mechanism, `can_capture` takes in the coordinate `node_to_place` and the current player whose placing it. The function loops through the array of adjacent nodes with tuples of coordinates from `generated_adj_nodes` of `node_to_place` and checks for two conditions, **horizontal capture** and **vertical capture**.



Suppose we are the player "`red`", the `node_to_place` is the red cross. This condition checks if the current and next nodes (blue pieces) are of the same colour, then find their matching adjacent node (excludes `node_to_place`). If the matching adjacent node (red piece) exists, capture is successful, and the two blue pieces (current and next) will then be added into captured array.

*Figure 3. side by side capture (horizontal capture)*



On the other hand, the second condition checks if the prev, current and next nodes matches the pattern of (blue, red, blue). If the pattern matches, the nodes can be captured, and the following two blue nodes will be added into the `captured` array.

*Figure 4. opposite capture (vertical capture)*

*where grey cells are the adjacent nodes of node_to_place to be visited and red cross is the node_to_place*

**Performance Analysis**

Let b = maximum branching factor, d = maximum depth and let e = time complexity for evaluation function
The Big-O time complexity for the minimax algorithm will be $O(b^d)$
This is derived from

$$(1 + b^1 + b^2 + \ldots + b^d) * e \qquad = b^d * e \qquad = \mathbf{O(b^d)}$$

The Big-O space complexity would be **O(bd).** This is because of depth-first exploration.
Using alpha-beta pruning will improvise Big-O time complexity to $O(b^{d/2})$.
On a size n board the branching factor of the game tree is $O(n^2)$.

The time complexity for evaluation function is dependent on the piece's position on the board.

**Other optimisations:**

***Applied Approaches***

***Depth***

The original minmax evaluation algorithm we decided on had a depth of 3. However, this increased the space and time complexity, exceeding the resource limits. Hence, we decided to change the depth to 2. This allowed us more leeway for resource limits.

***Swap Rule***

The evaluation function will implement a swap rule if the red node is placed in a swappable area. This allows player 2 a numerical advantage. The minimax algorithm takes this into account and creates a negative value if the player2 is able to swap player1's node.

***Taking threats into account***

If there is an available capture on the board, the algorithm will prioritise avoiding that outcome. This is due to the importance of retaining the goal path.

**Possible approaches to improve the evaluation function:**

A few reports were discovered while we were brainstorming ways to improve the efficiency and quality of the evaluation function. We also attempted at implementing these techniques, but failed to do implement these strategies due to lack of technical skills. We also decided against implementing some of these techniques as we understood that it might lead to over-consumption of memory and space (due to the time and space constraints).

These ideas were sampled from Chalup, Mellor, and Rosamond's (2005) report on writing an AI for the Hex project.

***Attempted Approaches:***

***Dijkstra's***

One of the simplest connectivity measure we can use is to count number of pieces required to connect p's edge piece. This can be counted using Dijkstra's algorithm, to find the shortest path to the goal nodes.

**Data Structures & Implementation**

2 classes are introduced: Player and Board

| Class | Description |
|---|---|
| Hex | Contains information regarding a particular node in the graph. |
| Graphs | Weighted graph, sets up the board state with empty/player nodes. |

*This approach was attempted and put into the other_agents directory*

***Other Approaches:***

***KCL equations***

This approach is more physics and math related for our level of study.
It requires the algorithm to model the coordinates as an electrical resistance circuit (which accounts for all alternative paths and their costs). A connectivity value fWor the coordinate is obtained by applying an electrical voltage V across the edge pieces and calculating the total resistance, $R_p(s)$, of the resulting circuit. The circuit is a complicated one, but $R_p(s)$ can be calculated by application of Kirchoff's Current Law (KCL).

"A set of KCL equations are written out for each node (given the usual assumption that the direction of current through the resistors is consistent across the equations), and solved simultaneously to yield the voltages at each node. Since all the individual voltages and resistances are known, the total current leaving the circuit can be calculated, which in turn yields a value for the equivalent resistance Rp(s) of the whole circuit. Finally, the value of Rp(s) provides the connectivity measure for the position." (Chalup et al., 2005)

This idea is implemented by Hexy, one of the strongest Hex AI programs invented around 2001.

***Minimising the branching factor***

There are a few pruning methods that could have been implemented, (e.g. (α-β)–pruning and beam search(forward pruning), and transposition tables, which cache the value of a position for lookup (but increases space complexity).

There is another method suggested by Chalup et al.(2005), called connection templates. They are patterns of cells that ensures a certain amount of connectivity between two cells called terminal points.

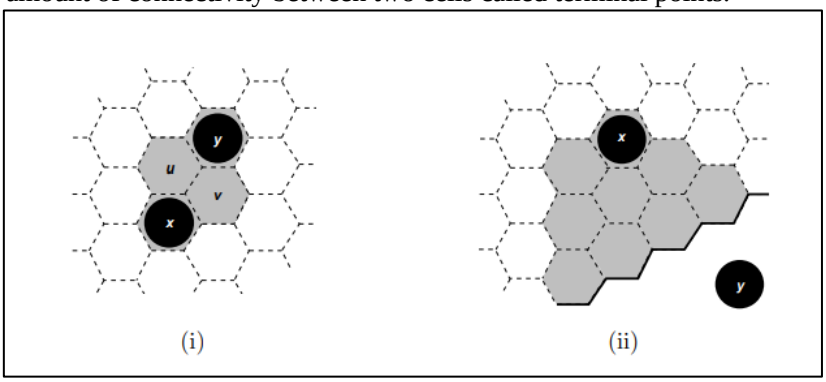



*Figure 5: (i) two-bridge template (ii) edge template*

*"Consider the two-bridge pattern: if white plays at u then black responds with v to complete the connection between x and y; alternatively if white's play is v then black can respond with u; hence the outcome of a two-bridge is predictable two moves ahead."*

In summary, it uses deep analysis with **patterns** instead of using metrics to look ahead. Different connection templates combine to create a "prediction board" that helps the program look ahead of their opponents moves. A weakness of this implementation is that it does not confirm that the nodes form a connection between terminal nodes.

*"A more sophisticated modification would distinguish between cells that are neighbours in the normal sense and cells that are neighbours because they are connected via the terminal points of a connection template, and assign perhaps a slightly higher two-distance value in the latter case."* (Chalup et al., 2005)

**References**


- Chalup, S., Mellor, D. and Rosamond, F., 2005. The machine intelligence Hex project. Computer Science Education, 15(4), pp.245-273.
- Galli, K., 2019. Connect4-Python/connect4_with_ai.py at master · KeithGalli/Connect4-Python. [online] GitHub. Available at: <https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py> [Accessed 11 May 2022].