Team-Ion Report

Assignment 2 : minimax agent for Cachex game
1085394 :- Sri Manokaran
1054297 :- Yukash Sivaraj

Describe your approach

Important information:

Our agent uses several functions to determine the final coordinate that will be placed on the board.These functions happen in a sequence so that in all cases there is a possible move that can be done. For instance, if our agent's current path is blocked we have a function to deal with that case. Instead of using a weighted calculation for minimax we set the evaluation function as described below and use other methods to decide the final move if the evaluation function is not conclusive.

For board structure we used the board structure that was provided as part of the skeleton code for the referee.

General rules in our decision making algorithm

1. **Next possible move:** for all functions, the next possible move was always considered as the neighbours of the last move that was played by the player. This was done to eliminate any redundant calculations done by looking at all nodes on the board. **(LIMITATION)**
    a. The only exception to this was the check_capture function, which is explained later.
2. **Returning False:** Since we are only looking at neighbouring nodes, if there is no possible move returned by one of the aforementioned functions (because there might be no neighbouring nodes), then we return "False". By returning this, we can determine which function was responsible for placing the piece on the board.

Search algorithm explanation:

The main logic of the game is contained within action(), which was provided as part of the skeleton code.

As all our agent's moves are looking at neighbours of the last move of our agent, we:

(1) Check if the next possible move is a coordinate that results in victory, if so place the piece.

(2) If not then run the minimax function for all neighbouring nodes of our agent's last move and check which node has the most positive effect on our agent (highest utility value) assuming the opponent will always play to our disadvantage.
(3) If **minimax is inconclusive (no coordinate returned)** then check if you can **capture a piece around the last move to make some space to make a path**
(4) If all of this is not possible then finally back track the path we have made to find an open spot to place the piece.

Function explanations:

(1) **final_coordinate()**
The final coordinate function is used to determine whether there is a possible neighbour to the last move by our agent that would lead to the victory of the game. If so, place this move leading to a guaranteed victory. This is done as a check at the **beginning of each turn** by looking at the path that we have already traversed and the neighbours of the last move.

(2)

minimax_decision() and minimax_value()
If the next possible move doesn't result in a victory (i.e we are still moving towards the goal), then call minimax_decision().

The unoccupied (coordinates that don't already contain pieces) neighbouring nodes of our agent's previous move are passed into minimax_decision() along with the board (state of the current board).

minimax_decision() returns the node that we are going to place on the board or it returns False if inconclusive (no new possible moves can be made from our agent's previous move).

minimax_decision() goes through each unoccupied neighbour and picks the neighbours with the highest utility value. The utility value is calculated using minimax_value().

minimax_value() returns the utility value for our current move, computed by the evaluation function: number of our agent's pieces - number of opponent's pieces. It does this by comparing all of the opponent's predicted moves with our agent's move. The predicted moves are based on the assumption that the opponent will play their move adjacent to our agent's move. We chose to implement this type of prediction as we think that the opponent will attempt to block/place a piece close to ours for capturing opportunities **(LIMITATION)**. The function then computes the utility value for each scenario and returns it.

It is possible that there are multiple neighbours with the same utility value, so in order to pick a move, minimax_decision() finds the move which has the most number of neighbouring free nodes.

We chose this strategy because our agent would have more open neighbours available for the next move, which increases the chances of our path going towards the goal and also increases the chances of not being partially blocked by the opponent's pieces.

(3) Modifications to the minimax algorithm:
check_capture()
This function is called if minimax_decision() returns False, meaning there are no new possible moves that could be made by our agent. In this case, we check if a capture could be made anywhere that is 2 depth levels away from our last move. The function looks at our last move's neighbour's neighbours that are unoccupied. We then calculate the utility value for each of these possible moves and pick the move with the highest move. If a move with a higher utility value than the last move is chosen, then a capture has occurred (since we have more pieces) and we return this move. If a move could not be found, False is returned.

(4) back_track()
The backtrack function is a last resort function if there is no minimax move, a capture move or a final coordinate move. This would backtrack from the last move our agent made to all possible moves to check whether there is a neighbouring opening to place a piece. This very rarely happens but is a surety that we keep playing the game and can reconfigure and play using the other strategies discussed above for the next move.

Features of evaluation function

The minimax function is dependent on an evaluation function.

Evaluation function: (Number of our nodes in the cachex board - the number of nodes the opponent has on the board)

Our strategic motivation behind this evaluation function is that our agent will avoid moves that may be captured by our opponent, since this will give us a lower utility value compared to a move that doesn't lead to being captured, likewise, our agent will capture one of our opponent's pieces, since doing so will give a higher utility value. The only exception is when the final move will lead to a win, in which case final_coordinate() will be called.

Performance evaluation:

Throughout the development phase of the agent we had developed multiple versions of the current algorithm before ending up with the final version. Initially, we developed a trivial solution with a random agent, then an agent which simply focused on reaching the end from the start without considering opponent moves, then finally the agent which incorporated minimax. We judged the overall effectiveness by running the code against each agent and averaging the score by how many games each agent won against each other. We decided on the agent with the highest score, which was the minimax agent.

Limitations and Possible Enhancements

A major limitation of the strategy is that the minimax function only looks at the neighbours of the last move and the neighbours of neighbours of the last move for the opponent move.
Which means the current implementation of the strategy only looks one move ahead. This could be further enhanced by using an extensive minimax search to look multiple moves ahead, possibly by looking through all coordinates of the board. It naturally follows that this enhancement will use so much more space and increase the time complexity.

To mitigate this, we can extend the idea with alpha-beta pruning (discussed in lectures) which decreases the branch the number of branches to evaluate. We also considered combining a search algorithm like dijkstra’s algorithm with the minimax function. This way we can give a weighting to our evaluation function, by assessing whether a capture is more important in a particular scenario rather than moving towards the goal.

Supporting work

The team members manually played through many games with each other first, in order to check what strategies we were implementing as humans and how we could potentially transfer these ideas to our agent.