Artificial Intelligence: COMP300024 Project B Report

Mukul Chodhary 1172562, Aimee Liang 1090840
(Dated: May 11, 2022)

Our AI agent *Disappointment* takes advantage of the key features of the CACHEX games, capturing system and forming a shortest path between two sides and utilises adversarial search techniques such as Minimax to think increasing number moves ahead as game progresses. We have also implemented some pruning strategies such as Alpha-beta pruning, Forward Pruning and time restrictions on a Minimax search time.

I. INITIAL PLANS AND TAKEAWAY

Our initial plan was to utilise the power of gradient descent and TD leaf (λ) to make a time efficient and optimal agent. Through the development cycle, the implementation of TD leaf (λ) turned out to be much harder than expected and we were disappointed. So we decided to change our strategy to use Minimax search. The time constraint was the biggest enemy of Minimax, and we had issues with it for a long time. We also considered taking a probabilistic reasoning approach.

We will discuss our final strategic approach, optimisations, modifications and how we drew inspirations from gradient descent and TD leaf (λ) to come with a good evaluation function in the following sections.

II. STRUCTURE

1 [Early Game]

- **Red**
  For the starting move, our algorithm uses a small book of opening moves to randomly pick a starting move. This is to prevent agent from recomputing moves at the start of the game which is very time-expensive. An agent is only allowed n^2 seconds per game, where n is the size of the board. We are randomly choosing the moves from the starting move space as it makes two consecutive games more exciting.
- **Blue** Blue player (with the assumption of being the second player) will be given the choice to perform 'STEAL' move. From extensive research on HEX game, we concluded that a Blue's first move should always be 'STEAL' and second move should be on the shortest path. Blue has a clear advantage by performing move 'STEAL'. Because this not only gives the Blue player one more piece on the board, but also disrupts opponent's short term game strategy [1][2] [3]. We found that sometimes, the 'STEAL' move will not always result in an optimum move for the Blue player, and it can increase time required to find the next move (need to explore more of move space in Minimax). To avoid this overhead, we compute a move in the current shortest projected path using A* search (see 8 in section IV).

2 [Mid Game to Late Game]

There are no distinction between how the *Disappointment* agent handles Red player and Blue player in Mid to Late game.A greedy next move approach is first taken with some consideration to opponent's current state in the game. This move is then evaluated over two features: if the current move moves closer to the goal compared to all previous moves, and if number of captures this move can or will be able to perform (see 9 for detailed explaination). These moves are considered greedy as they don't completely consider opponent's response which sometimes resulted in 'bad moves'. Therefore, after many runs of the game, we picked a stable threshold which effectively split the relatively good moves from bad moves. If the evaluation of the greedy move is below this threshold, we utilise Minimax search with increasing depth to find an optimum move (see 12 in section IV for in depth detail regarding Minimax).

The depth of the Minimax is increased for every total number of pieces on the board is a multiple of 17^{th}, this is parameter was found by trial and error and can be further optimised through hyper-tuning. We decided to go with '17' as it ensured us there are enough pieces on the board as well as the move space is relatively smaller. This approach allow us to consider number of player and opponent's pieces on the board with respect to board size instead of number of turns. In earlier versions, number of player's turn was used. This version was later discarded as number of turns does not accurately represent the game state and number of pieces on the board could differ drastically due to capture system. We also came up with keeping track of number of player's pieces on board to distinguish different stages of game and player's status with possibilities to apply different strategies accordingly. Both of these approaches represent the state of the game, and in most cases number of pieces would be a better parameter to go for this. However, future versions would involve testing these two approaches over larger games. We were not able to do such due to limited time and computer resources.

3 [End Game]

Towards the end of the game, breaking opponents path, blocking their path and quickly connecting our path should be prioritized in our strategy as we are short on time at this time. Moreover, after extensive comparisons of games we found the result presented in Minimax and our greedy approach were similar. The greedy

moves, are specialised to achieve the mentioned features. Therefore we decided to completely utilise these moves instead of Minimax due to pressing time and being in late stage of the game. These greedy moves can be found in less than 0.01 seconds, thus making them ideal for later games where time is really precious.

III. MODULES AND CLASSES

The directory can be broken down as follows (important methods shown):

```
AI_PB
├── referee
├── disappointment
│   ├── __init__.py
│   ├── path_cost_weights.py
│   ├── path_search.py
│   │   ├── Graph
│   │   └── a_star_search()
│   └── player.py
│       └── player
│           ├── action()
│           ├── turn()
│           ├── game_state_eval()
│           ├── alpha_beta()
│           │   ├── move_space()
│           │   ├── max_value()
│           │   ├── min_value()
│           │   └── mini_max_eval()
│           └── get_next_move()
│               ├── find_path_move()
│               └── find_capture_move()
└── emotionaldamage
    ├── __init__.py
    └── player.py
```

4 [**player.py**] Implements the adversarial search algorithms (Minimax with pruning) as well as methods to maintain internal game state.

5 [**path_search.py**] Implements graph class and A* search algorithm and supporting functions. Graph class is used to keep track of board internally. This is a modified version of our Project-A submission.

6 [**emotionaldamage**] A random agent used for testing purpose.

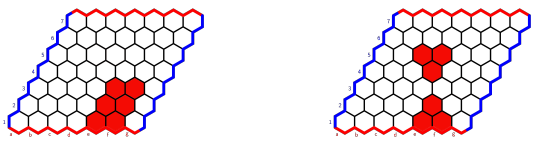
IV. NEXT_MOVE AND HEURISTICS

This section presents the heuristics used throughout the agent logic.

7 [**Greedy Moves**]

- **Shortest Path Move**
  The shortest path is evaluated from all start to end available positions. For red player start-end positions refer to all available cells along the top and bottom of the board, and for blue player these are all available spots along the left and right lines of the board respectively. This means that A* is run at most n times for each starting position. This means time complexity to find a path using A* is $n \times O\left((b^*)^d \log\left((b^*)^d\right)\right)$, where $b^* = b^\epsilon$ (see 8 for more details regarding A*).
  A* search method in path_search.py provides us with cost and path where path with the smallest cost is stored. This minimal cost path is processed to find the next unoccupied cell (our next shortest path move) then cost of the path will be used to evaluate the move. Greedy move evaluation is further explained in 9. Essentially, the move evaluation is a combination of *Path Progress Contribution* and *Capture Potential*.

- **Capture Move**
  Capture Move is a specialised move inspired by the capture system where all current pieces are searched through to see if there is any capture available, i.e. if there is any cell where putting a piece will result in capturing opponent's unit or prevent the opponent from making moves which leads to capturing player's units. The total number of captures possible are added together and are used to evaluate the Capture Potential. The best capture move with highest capture potential is then applied to the copy of internal game-state to figure out its Path Progress Contribution.
  The total move evaluation is a combination of path progress contribution and capture potential(see 9 for more details).
  If the total move evaluation is lower than the threshold to check if there are any defensive moves possible which improves are players position in long run. Some examples would be creating a double wall to prevent opponent capturing pieces as well as making a making a move in triangle orientation (Figure 1).



(a) Double Wall

(b) Triplets / Moves in Triangle Orientation

FIG. 1. Defensive Move Examples

8 [**Weighted A* Justification and Inadmissible Heuristic**]

In Project A we discovered that A* has time complexity of $O\left((b^\epsilon)^d \log\left((b^\epsilon)^d\right)\right)$, and $\epsilon \equiv (h^* - h)/h^*$, where h is the expected cost and h^* is the actual cost from start to

goal [4]. On average, this reduced to $O\left(1.17^d \log 1.17^d\right)$ with an admissible heuristic.

In this project we decided to use weighted A* with an inadmissible heuristic as we gave more importance to number of nodes explored as well as the speed of the search. Weighted A* fits both of these criteria [5] nicely but it does provide a sub-optimal. However, the sub-optimal is acceptable as the agent only acts on one move in the path and the game state is updated in every move. Therefore, even if we had an optimal path the game state would have been changed and a new path must be recomputed every turn. Thus, we decided to use faster and less memory intensive weighted A*.

9 **[Greedy Move Evaluation]** As mentioned in previous section, total move evaluation is the combination of path progress contribution and capture potential which will be discussed in detail below.

- **Path Progress Contribution** Path Progress Contribution is essentially a metric to evaluate how much closer will this move takes the agent in making a complete path in comparison to all of previous moves made. Game-state keeps track of all of previous moves made by the agent as well as the opponent.
  Let **A** be the vector containing all of previous moves cost. Let **B** a vector of length **len**(A) such that all elements are 1. Let 'c', a scalar, be the cost of the current path move.Let L = **len**(C), and let L^{th} element of A, A_L be cost of most recent move made, i.e. move cost are appened to the end of the vector A.
  Then the cost difference vector, $\mathbf{C} = \mathbf{A} - c * \mathbf{B}$
  Relative path cost = $\sum_{k=0}^{L} \frac{C_k}{2^{L-k}}$. This is done to figure out how well the current move is doing compared to previous moves, with geometric series as weight so more recent moves are weighted much more than old moves.
  Path progress contribution = sigmoid(relative path cost) * relative path cost.

$$\textbf{sigmoid} = \frac{1}{(1 + \exp(-c_1(x - c_2)))}$$

  $c_1 = 0.4, c_2 = 0$ and sigmoid $\epsilon(0, 1)$.
  Path progress was calculated this way so 100% of relative path cost contributes to the evaluation only when that number is high, and the contribution of relative path cost to total move evaluation decreases as the relative path cost decreases.
  This approach was heavily inspired by some of the features of gradient descent as well as TD leaf Lambda which looks at previous moves for evaluation and weighting.

- **Capture Potential**

$$\mathbf{C_{potential}} = a \times \tanh\left(c1 * (\mathbf{nCaptures} - c2)\right)$$

  where a = 4.1, c1 = 0.4, c2 = 0 and **nCaptures** represent number of captures .
  These parameters were chosen to give capture potential of 1 a score above the the cut off score. The hyperbolic tangent function was used to scale number of captures non linearly. Hyperbolic tangent was chosen instead of a sigmoid because a sigmoid function has a point of inflection, meaning for first half its concave up and slowly increases. This is not ideal for our case as we need a rapid increase in scores as number of captures increases by 1. Hyperbolic tangent is concave down and fits the wanted characteristics.

10 **[Game State Evaluation]**

Game State Evaluation looks at only "how close is my agent in making a complete path compared to that of opponent's" feature and is called in *turn()* of the agent. This is mainly used to keep track of all costs for both player and the opponent, which is used in Path Progress Contribution Calculations.

11 **[Minimax with Alpha-beta Pruning]**

In order to make our Minimax tree search more efficient, we optimized it by adopting Alpha-beta Pruning to reduce our search space. To further find useful search space we limited our search space to 1st degree neighbours of the placed units (both agents, and opponents). As the total number of empty spaces are very large at the start of the game and these can significantly increase the search time.

The search space is a combination of opponent's pieces' 1st degree neighbours and agent's pieces 1st degree neighbors in this order. This allows Minimax to effectively explore all the defensive moves before it can explore other strategies.

Forward Pruning is strategically used to look one move ahead to find terminal state. If terminal state is reached from this move, the player will stop the search here. This allows us to exit the Minimax search earlier which boosts the agent's performance.

As there are limitations on the total time given to the agent in a game, we have decided to limit the amount of time agents spend on performing Minimax search by introducing a time threshold - if the agent goes over the time threshold, Minimax returns the current best move found. This threshold is set to n+n//2, as we found this to work well in most cases. The perimeter can be optimised much more using machine learning techniques, which can be explored in future discussions.

To ensure our agent has enough time to finish the game, we let the agent use greedy moves if the total time spent by agents goes over total time - one Minimax move time, i.e. $t_{\text{go-greedy}} = n^2 - (n + n//2)$. If $t_{agent} > t_{\text{go-greedy}}$, agent only uses moves listed in 7.

Agent also evaluates move evaluation based on greedy evaluation mentioned above, this is then used to compare if the move calculated by Minimax is better than greedy move. If the Minimax move has a lower value than greedy move, we use the greedy move. This check is performed to tackle against greedy nature of Minimax induced by

time restrictions.

The depth of the Minimax is designed to increase dynamically with the total number of pieces on board. This parameter is currently set to increase by 1 when the max number of pieces on the board so far becomes a multiple of 17.

12 **[Minimax Evaluation]**

Minimax evaluation is the sum of game state evaluation (see 10) and difference between number of agent and opponent's pieces on the board. The evaluation function also looks at if the agent is about to finish making its path, or the opponent is and gives them a really high evaluation and low evaluation respectively, i.e. some_big_number and -some_big_number.

V. OPTIMISATIONS

The following is a list of overview of optimisations used in our design(some of them are extensively elaborated in previous section):

- **Starting Position** Using book-learning approach will improve our time and space complexity with consistent strong opening strategy
- **Re-use of Part A**
  We reused and modified our Part-A graph class and weighted A* search algorithm in Part B.
- **Optimised Greedy moves**
  We evaluated to important moves based on key features of the game CACHEX, to reduce time spent in Minimax search.
- **Alpha-beta Pruning in Minimax Search**
  We implemented Alpha beta search, forward pruning, and end game-state evaluation to prune the search tree and quickly find the move. The Minimax evaluation function also uses key game features such as path completion, captures, and total number of agent's units vs. opponents unit. The move space was restricted to 1st degree neighbours and the depth of the Minimax increased as the total number of pieces on the board increased.
- **Time Restrictions per Move**
  There is no time restriction on greedy move, as they can be evaluated under 0.01 seconds.
  Time restriction on Minimax move is $n + n//2$, where n is the size of board.
  The time threshold when the agent goes completely greedy is $t_{\text{go-greedy}} = n^2 - (n + n//2)$.
- **Time Optimisations and Restrictions**
  We carefully set time restrictions to Minimax search, and if the $t_{\text{go\_greedy}}$ was reached agent only used optimised greedy moves. To ensure the moves out of Minimax search are actually better than greedy moves, this is not always true due to time restrictions and preemptively stopping the search, we compared it with the greedy move and returned the best move.
- **Space**
  We handled memory and kept max usage of memory less than 1 MB with success.

VI. AGENT APPROACH AND PERFORMANCE

13 **[Results]**

Below display the outcomes against potentially suboptimal opponents. Using bash scripts to execute the program, we created a graph below showing the results after 200 matches of different board sizes. For efficiency, we chose 4 board sizes that are representative - 3, 5, 7 and 8. We set agent *Disappointment* to be Red player, and agent *EmotionalDamage* to be Blue player.

Our agent Disappointment dominated the matches against EmotionalDamage, who is a random player, with a very positive looking percentage which is detailed below:

- Running on board size 3, Red won 98 out of 100 matches, success rate 98%.
- Running on board size 5, Red has a 99% success rate out of 100 matches.
- The success rate of Red is 100% on a size 7 board out of 200 matches.
- Running on board size 8, Red has 100% success rate out of 100 matches.

This result is expected as for larger board sizes mini max has more allocated time per move, therefore moves picked are more optimal. The trend should continue as board size continues, and agent *Disappointment* will be able to play more optimally using mini max algorithm

FIG. 2. Comparison between Red and Blue player after 100 and 200 matches, with Red being Disappointment and Blue being EmotionalDamage.

14 **[Future Considerations]** The following are areas of improvements/open to enhancements:

- Combination use of Reinforcement Learning with the technique of TD Leaf.
- Parallelism in Alpha-beta Pruning, using API (e.g. Open MP) that supports implementation of Alpha-beta on shared memory. [6]

· Reordering on child nodes for faster search

[1] C. D. of Mathematics, HEX.

[2] L. Jing, HEX.

[3] G. Bodwin and O. Grossman, *Strategy-Stealing is Non-Constructive*, Tech. Rep. 185 (Electronic Colloquium on Computational Complexity, 2019).

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. (Prentice Hall, 2010) pp. 97–98.

[5] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. (Prentice Hall, 2010) pp. 90–98.

[6] S. Mandadi, S. Vijayakumar, and T. B., Implementation of sequential and parallel alpha-beta pruning algorithm, International Journal of Innovations in Engineering and Technology **7**, 98 (2020).