A Cachex Agent Based on Adversarial Search

Team: ELDEN_KING
Jiahao Chen, Zhiquan Lai

I. Introduction

The overall goal of this report is to present our development of a Cachex agent which is implemented based on the Minimax algorithm. The remainder of this report is organized as follows. The mechanism and structure of our agent are described in Section II, which is the foundation of the whole system. Then in Section III, a couple of helper functions are contained, which pave the way for the next section. Implementation of adversarial search techniques, as well as our discoveries and improvements, is discussed in Section IV. Next in Section V, we evaluate the performance of our agent from multiple dimensions. Finally, conclusion and future improvements are set in Section VI.

II. System Structure and Mechanism

A. Overall Structure

In order to increase the reusability and extendibility of our system, object-oriented programming is applied in Part A of this project. We generally follow the same structure and make improvements based on the new features in part B. Figure 1 shows the basic structure of the system, including some important methods.

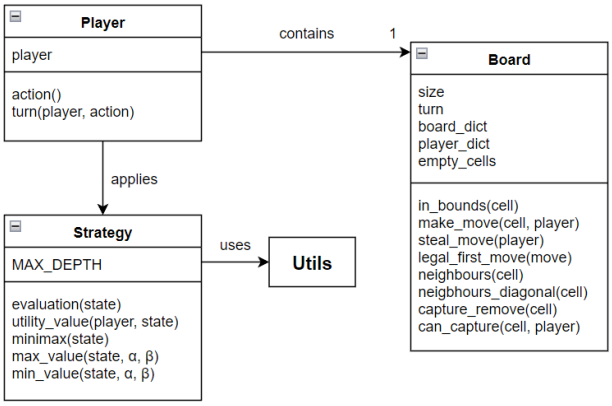


Figure 1. System structure

A board class is initialized for the player when the game is started by the referee, which keeps updating the game states. During the game, the player makes decisions of its actions based on information from the board, using the methods from the strategy module which will be discussed in section IV. The utils module contains all the supporting work that will be shown in the next section, which helps the agent formulate game strategies. Next, important mechanisms in the board class will be explained.

B. Board Mechanism

The board is of great importance since it records all the information that the agent requires and guarantees that each move is regarded valid by the referee. When the board is initialized, it mainly records 5 elements. Firstly, the size of the board is saved as a static constant, which helps to check if a cell is in bounds. Also, we find that the size has a huge impact on the efficiency of our agent, and it will be discussed later. Next, the board updates the current number of turns since we have some special rules in the first and second turns. Finally, it maintains two dictionaries and a list to keep updating status of all the cells on the board. Figure 2 shows an example of how the board is recorded if the player is red.

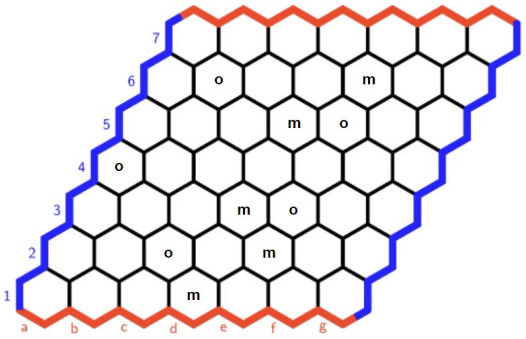


Figure 2. Board example

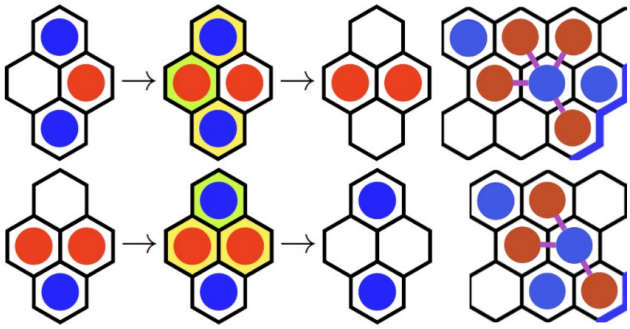


Figure 3. Types of capture

We place two flags representing two sides, “m” for my side and “o” for the opponent. The color of each side is saved at start in a dictionary (player_dict) with colors as keys and flags as values. After each round, the new cell being occupied is added as a key in the board dictionary with its value equals to the corresponding side and it is removed from the list of empty cells. The reason why we choose to use dictionary is that accessing the status of a cell by the key is fast (O(1)) and it helps us to program in a more intuitive way. In figure 2, all the occupied cells are keys in the board dictionary with values of either “m” or “n”, and the rest are kept in the empty cells so

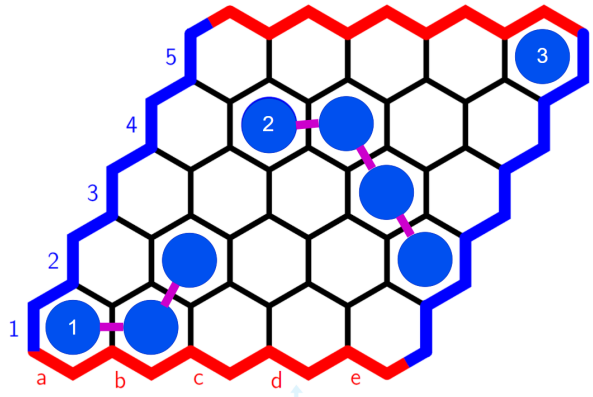
that we do not need to recalculate all the available actions in each turn and the agent can get fast access to them.

One of the most important elements in the game is the capture rule, which affects our choice of the utility function to a huge extent. To make sure the agent can find correct captures, several tests are done, and we designed a bug-free algorithm. There are two types of captures as left part of figure 3 shows, the Manhattan distance from the cell that forms the capture to the cell placed before is either 1 or 2, and the cells being captured are all 1 distance from the cells that captures them. Following this logic, the algorithm attempts to find if there are possible captures of both types. Surprisingly, there exists situations that one move can perform more than one capture as right part of figure 3 depicts, which is a huge contribution to winning the game.

III. Support Work

In order to construct a powerful evaluation function for the adversarial search, we select a couple of features based on our understanding of this game, aka empiricism. To accurately measure these features, we come up with different algorithms.

A. *Maximum length of consecutive pieces*




```
while cells (all cells occupied by me) not empty:
    pop cells[0]
    add the poped cell to a queue as head of a new path
    while queue not empty:
        pop queue[0]
        add occupied adjacent cells to queue
        remove occupied adjacent cells from cells
    update the max length of path
return max length
```

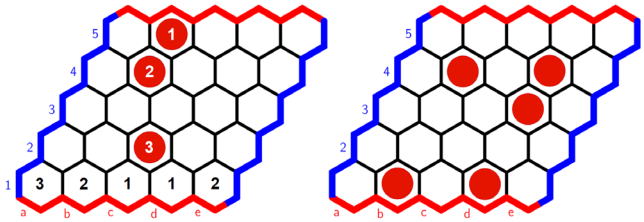
Time complexity: $O(H * (6^N))$
Each cell normally has 6 neighbours
H: Number of heads of path
N: Number of cells along each path

Figure 4. Max length of path

To begin with, we consider the max length of consecutive pieces is a significant feature because it means that the player has a long path, which tends to increase the probability to win. To find the max path, we design an algorithm based on breath first search as the pseudocode in figure 4 shows. We apply BFS on each possible head to find the optimal solution. In reality, H and N cannot be large simultaneously under most cases because as the number of total cells increases, more cells are likely to be chained together so that H is small. If the cells occupied are sparse, the cost of BFS will be greatly reduced. For the example in figure 4, H is 3 and N equals 3, 4 and 1 for three paths respectively.

B. *Find best start and goal positions*

Another important feature is the minimum cost currently to win, which is similar to what we have completed in Part A. However, we have no ideas about the start and goal before applying the A* search because all cells on the boundary can be chose. Therefore, we establish an algorithm that tries to find the optimal positions of start and goal so that we only need to apply A* search one time to find the min cost. According to the rules, the red player needs to build a path vertically from bottom to top and the blue player needs to construct a horizontal one. Let us take the red side as an example and assume that the path starts from bottom and ends at the top border. As presented in figure 5, we mainly formulated 3 rules to find start and goal. Looking at the left one first, red 1 is already at the top and we can take it as a possible goal state. Red 3 is the one nearest the bottom, and we can select the empty cell on the bottom which is nearest to it. The Manhattan distance from each bottom cell to red 3 is marked. In case some of them are occupied by the blue player, for example (1, c), the algorithm will then choose (1, d) as the next optimal starting cell. If (1, d) is also not available, (1, b) will be taken, and so on. The result will be None if there is no available cell.



Rule 1: Cell is on the boundary, select it as a start/goal
Rule 2: Select the cell that is nearest to the border, find optimal start/goal
Rule 3: In case of a tie, select the line with more cells

Figure 5. Find start and goal

One of the special cases is presented on the right part, which we have multiple possible start positions. We decide to select the cell which has more pieces in the same column (row for blue player) as it leads to less cost to win. For the example in figure 5, the cost to win is 3 for column b and 2 for column d and it is apparently that the latter one is a better choice.

The overall time complexity to find start and goal is within $O(n^2)$ on average with n representing the size of board. The best case is O(1) when there exist pieces on the boundary. Since n is bounded between 3 to 15, the complexity has little impact on the whole system even in the worst case. However, after a number of tests, we find that the algorithm may returns a suboptimal result since the board is ever-changing. Therefore, we regard it as a relatively optimal but fuzzy result. It surely helps to find a good path but there may be a better one. To solve this problem, we make a few improvements in our A* search so that the min win cost is ensured to be optimal.

C. *Minimum cost to win the game*

This problem is similar to the challenge question in Part A since we can take advantage of existing pieces to build our path. In Part A, we designed a heuristic function h(n) = Manhattan distance - number of existing pieces, with the cost of using existing pieces as 0. If h(n) = Manhattan distance, the result may be suboptimal like the green path in figure 6, which we have presented in our Part A's report. Since the new h(n) takes the existing pieces into consideration, it cannot be greater than the true cost so that it is admissible. If the number of existing pieces exceeds the Manhattan distance, the algorithm becomes a uniform-cost search to find the optimal cost.

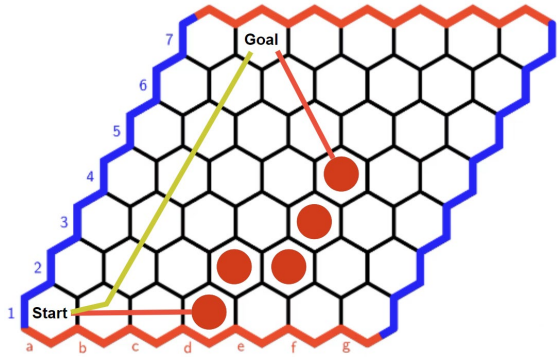


Figure 6. Example of suboptimal path

However, as we have mentioned above, positions of start and goal may be suboptimal and figure 7 provides an example. (1, a) is chosen as the start state while starting with (1, c) results in the least winning cost. To solve such issues, we make the following improvements.

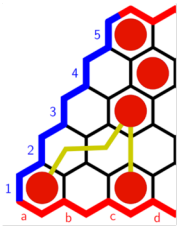


Figure 7. Example of suboptimal start

Firstly, if a cell that can be regarded as a start place is encountered, add it to the queue with cost equals 0 and no predecessor. Secondly, if a cell that can be a goal is visited, add it to a list of visited goals so that we can extract the goal with the minimum cost after the search. Finally, we add a weight of layer in f(x) to make cells at lower layers tend to have higher priority to be visited in case there might be a better solution. For the example in figure 7, though (1, c) is visited and be marked as another start cell with cost = 0 and f(x) = -1 (0+4-5), (4, b) still has a less f(x) = -2 (2+1-5) so that the optimal path is skipped. We add the layer value to f(x) so that once it finds a better solution in lower layers, it will first visit that cell and rebuild the optimal path. The layer value is the x axis for red players and y axis for blue players. It will not result in a suboptimal path because cells in the same layer maintains the same priority. This may arise a bit ambiguity for understanding but it indeed provides optimal solutions in all the test cases we have performed and greatly contributes to the later section. Figure 8 explains the algorithm visually.

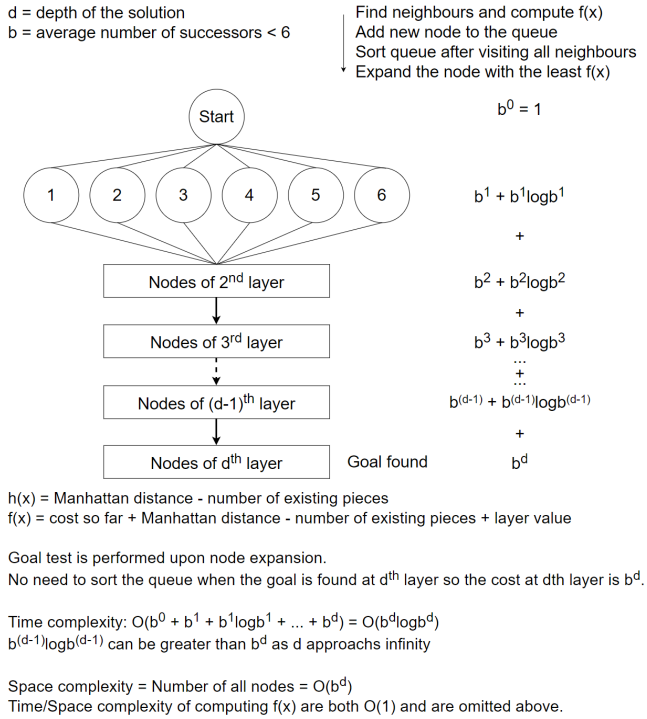


Figure 8. Analysis of algorithm finding min cost to win

IV. Adversarial Search

Next comes to the core part of our agent, adversarial search. It is mainly based on the Minimax algorithm and α - β pruning is applied to optimize the efficiency, which will not be the focus of this section since they have been discussed in lectures. In addition, we come up with several methods to improve the performance as well as efficiency, which will be explained in detail as follows.

A. *Utility function*

The key point of adversarial search is to establish a powerful utility function to evaluation each action. We select 6 features as depicted in table I with corresponding weights, which we consider are helpful to assess each action reasonably from multiple dimensions.

TABLE I. Features of Utility Function

ID	Feature	Weight
1	My number of occupied cells	1
2	Opponent's number of cells	-2
3	My max length of consecutive pieces	1
4	Opponent's max length of consecutive pieces	-2
5	Min cost for me to win	-2
6	Min cost for opponent to win	3

Weights of all features are generated through empiricism based on our discoveries from playing against other people and agents using different strategies. Features can be divided into 3 types: number of pieces, max length of consecutive pieces and min cost to win. They may be correlated to each other to some extent but meanwhile have their own propensities. For example, a higher positive weight for feature 3 may lead the agent to focus more on itself and less on opponent's status.

As soon as the agent can make actions by itself, we first play against it to test whether its decisions are rational. All features have the same absolute value of weight at first, and it generally can make good decisions. However, if we try our best to capture its pieces, it seems difficult for the agent to win because it cannot maintain a path, which hugely inspired us on the strategy to win. Simply speaking, we consider the best strategy of this game is to construct my own path without allowing the opponent to have a path to win. There is a saying that offense is the best defense, which is fully reflected in Cachex. Imagine that your opponent even does not give you an opportunity to build a path, and it looks ahead several steps to get you into this trouble, how can you defeat it? We have encountered lots of dilemmas when fighting it, for example, if we perform a capture, we will be captured in the next turn. The worst case is that wherever we place, we will be captured afterwards. Therefore, we set the largest weight for feature 6 to minimize the possibility for the opponent to maintain a path. Originally, we think that feature 1 and feature 2 may be redundant, but in case of a multiple captures as we have presented in Section II, these features may have more significance. Remember that feature 5 and feature 6 are None if no available path is found. We set the cost to twice the size of the board as a huge reward or penalty when computing the utility value. If the player wins by taking this move, it will return infinity meaning termination. While if the opponent wins by the move, negative infinity will be returned. Besides, more enhancements will be introduced later.

B. *Decisions in 1st and 2nd turns*

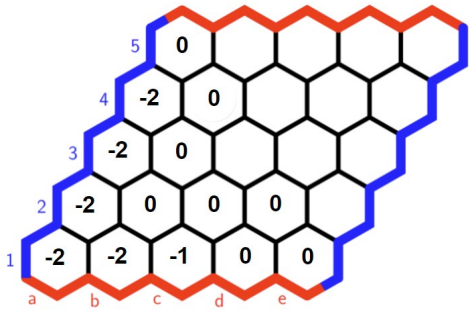


Figure 9. Utility values in 1st round

The swap rule is included to reduce first-mover advantage, which makes actions in 1st and 2nd rounds trickier. Let's first consider whether to steal in the second turn and our logic is simple. Assuming there is no swap rule, the agent normally performs minimax based on the opponent's move and it will obtain a good place. After that, we make the agent to apply the min value function on the opponent's first move, like the opponent has not make that move and the agent is now the first player. If the utility value of the first move is higher than the one we computed before as the second player, then the agent will choose to steal to get the first-mover advantage.

Now comes to the first round, how to avoid the opponent to make a good steal? Assuming that the opponent will definitely steal a good first move, it is rational to make a move that is relatively fair to both sides. Thus, we decide to take the cell which has a medium utility value. Utility values for cells in first turn are marked in figure 9 and only half of them is computed because of symmetry. Due to the depth limit we set for minimax and the limitation of information in first round, the numbers are close. We have done this job for boards of different sizes as well and the results seems similar, that the cells in the middle of the bottom tends to have a medium value. Considering that performing minimax in first round has a huge complexity because all cells are available, we simply let the agent to place the cell in the middle of the bottom according to the board size. This may not be rigorous, so we also have tested the final results of placing at different cells in first turn to see if the first mover really has an advantage. Tests are performed between agents with same strategies and the steal mechanism is removed. As figure 10 shows, cells with a red piece mean that player red first occupies this cell and finally wins while blue ones represent that blue player finally wins after red player occupies the cell in first round. It seems that occupying borders may be rational for first mover because cells on the boundary has less chance to be captured. But it may only make sense for our agent since we have a higher weight of capturing.

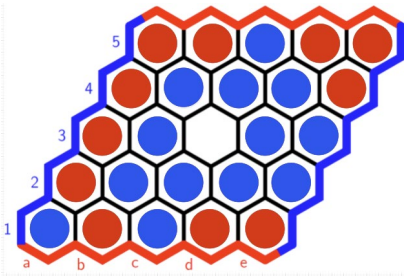


Figure 10. Check first-mover advantage based on our agent

C. *Efficiency optimization*

TABLE II. CONFUSION MATRIX FOR DENSE NET

Board size	Depth
[3, 5]	3
[6, 10]	2
[11, 15]	0 (Greedy)

Minimax and algorithms such as A* are time consuming, even if α - β pruning is applied. To optimize the efficiency, we first set a cut-off depth to force terminating the search as listed in table II, which should meet time and memory constraints on dimefox. As the board size increases, the time complexity will grow significantly, and we have to set stricter limits. Besides, we have a set a timer so that if the time limit is approached, the agent will force ending the search and return the optimal solution computed so far. Moreover, it is a huge waste of time to perform minimax on all the cells since many

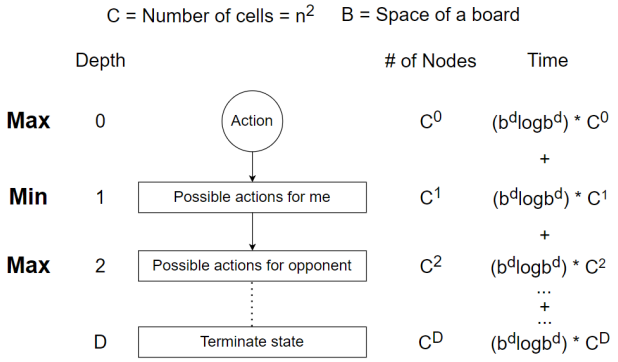
of them are apparently bad moves. Thus, we add a preprocessing step before minimax to filter out good moves, which is explained in figure 11. The branching factor of first layer in minimax is greatly reduced and the agent becomes more aggressive if opponent is about to win. We also have considered to add this logic to all layers, but it seems that the improvement is limited since it may have more chance to ignore a good move.

Evaluate current moves:
1. If I make this move, how much is my benefit 1
2. If opponent make this move, how much is my benefit 2
3. Eval value = benefit 1 - benefit 2
4. If I can win by taking this move, directly return it
5. If I may lose after opponent taking this move, first try capture then block
6. If opponent's min cost to win is less then 0,4 * board size, first try capture
7. Sort based on eval value in descending order and do minimax on first half
Note, benefit is computed using the utility function

Figure 11. Preprocess before minimax

V. Performance Evaluation

A. Time and space complexity



Utilty function -> $O(n^2 + 2 * (H * 6^N) + 2 * (b^d log b^d)) = O(b^d log b^d)$
1. Traverse board_dict to get my cells and opponent's cells => O(C)
2. Compute my max length of path and opponent's => $O(H * 6^N)$
3. Compute my min cost to win and opponent's => $O(b^d log b^d)$
See detailed analysis for 2 and 3 in above sections

Minimax
Get available actions => O(1)
Evaluate actions => $O(C * b^d log b^d)$
Sort actions and extract the first half => O(ClogC)
Max(min_value)

min_value
Termination test
Compute utility value => $O(b^d log b^d)$
Clone a board for deeper search => Space O(B)
Min(max_value)

max_value
Termination test
Compute utility value => $O(b^d log b^d)$
Clone a board for deeper search => Space O(B)
Max(min_value)

Total time complexity is bounded above by $(b^d log b^d) * C^D$
Cloning boards is significant to space complexity => $O(B * C^D)$

Figure 12. Overall complexity analysis

Figure 12 summarizes the complexity of algorithms in general. Obviously, the search depth (D) and length of a solution path (d) contribute the most. In fact, d is highly correlated to the board because we need to build a longer path in a larger board and that is the reason why large boards remarkably increase time. With α - β pruning and other optimizations we have done, the branching factor for minimax (C) can be efficiently reduced, which makes the agent possible to perform a deep search even if the board has a large size.

B. Playing against different opponents

To objectively evaluate the level of our agent, we find several opponents for it as table III shows. Firstly, we have not seen any case that the random strategy can win. For the greedy one, we simply apply the evaluation metrics that we use to preprocess actions as we mentioned above. The third one is similar to our agent except that its utility function only uses feature 5 and feature 6, which does not have a high focus on capturing. In addition to these agents, we also have invited some friends to play against it. The board sizes are mainly set as 8 and 9 and either side can go first. It turns out that our agent outperforms all kinds of opponents we have found.

TABLE III. BATTLE RECORDS

Opponent	Strategy	Result (1st)	Result (2nd)
Random	Random	100% win	100% win
Greedy	Current best move	win	win
Minimax	Simple Utility	win	win
Human	Brain	win	win

VI. Conclusion

In summary, we have developed an Cachex agent based on minimax and made effort to add enhancements. The result seems promising; however, it still has many problems, and we have huge room for improvement. For example, the opponents we select are not guaranteed to be optimal and our strategy is completely based on empiricism. Advanced methods such as reinforcement learning has been widely used in chess games and achieved satisfactory results. This project has given us an opportunity to delve into AI and will have a profound impact on our future research in this field.