AI-assignment2 Report

Bohao Liu 1036026
Yuxiang Xie 1060196

Our approach:

1. ***The algorithm we chose and the reason for our choice.***

   We choose the minimax algorithm plus the alpha-beta prune algorithm.

   First of all, this assignment is a game where two players act at the same time, and the goal of the game is to find a winnable goal state, so we abandoned traditional search algorithms such as depth-first search and breadth-first search and heuristic search algorithms such as a*.

   Secondly, this game is a deterministic perfect information game, which means we have complete control over the actions taken, and we can see all the information on the board, while the Minimax algorithm is a perfect play algorithm for such a game. Besides, the Minimax algorithm is complete and optimal, so we chose the Minimax as our search algorithm.

   Besides, in order to reduce the time complexity of minimax, we also use the alpha-beta pruning algorithm to improve the performance of our program.

2. ***Modifications to the minimax algorithm***

   The time complexity of Minimax is b^m. For this game, Although m=4, b can be a very large value. In addition, it is difficult to find the “perfect ordering” to reduce the time complexity to b^2, so alpha-beta pruning alone is not enough. Therefore, in order to improve performance and reduce the time required by the algorithm, in addition to alpha-beta pruning, we also designed a series of pruning methods. For details, see the Other Aspects section.

3. ***How have we handled the simultaneous-play nature of this game?***

   First, we fix the depth of the algorithm to a multiple of 2.

Secondly, in our minimax algorithm, we do not update the board immediately after max selects an action, but pass the action downward. After min also selects an action, we call the update function designed by us and pass two actions at the same time to update a deep copy of the board state.

4. ***Features of our evaluation function.***

   We use the double loop method to traverse our tokens and the opponent's tokens, so as to calculate the distance between each of our tokens and each opponent's token. When our token can defeat the opponent's token, the value will increase by a value that is inversely proportional to the distance; when our token is defeated by the opponent (for example, our token is s, and the opponent's token is r), the value will decrease by a value that is inversely proportional to the distance.

   In addition, we use the number of dead tokens as a parameter. When the number of our dead tokens increases, the estimated value of this state will decrease by a very large value (we set it to 10000). While the number of opponents‘ dead tokens increases, this estimate will increase by a very large value to ensure that the type of dead tokens is the biggest factor affecting the estimated value.

   To sum up, in our evaluation function, the type of new dead tokens is the factor that has the greatest impact on the valuation. When one of our tokens dies, the valuation is greatly reduced, while the opponent token dies, the valuation is greatly increased. Besides, the more tokens we can defeat the opponent, the greater the value, and the more tokens the opponent can defeat us, the lower the value. For a token pair that we can defeat the opponent, the greater the distance, the smaller the increase in value, for a token pair, that we will be defeated by the opponent,the greater the distance, the smaller the decrease in value.

Performance evaluation

1. ***How do we judge our program’s performance?***

   We use three factors to evaluate the performance of the program, the time required for the program to return to the selected action, the rationality of the program's choice of action, and the winning rate of our program.

First of all, we believe that running time is an important factor in evaluating program performance. In order to improve the performance of our program, we have designed a series of pruning functions to reduce the time required to select an action. So, now our average time per game is around the 30s.

Secondly, we competed with ourselves many times during the development process. In each game, we check the actions of each round to ensure that our choices are appropriate. Once we find inappropriate behavior, we immediately analyze the reasons and correct them.

Besides, the win rate of our program is also very important. We tested our program multiple times by competing with some other students and tutors through the battleground, then modified the program according to the results. After many improvements, our program finally reached the current performance: When the opponent is a greedy algorithm or a random algorithm, our winning rate is almost 100%. When the opponent is another student, our failure rate in the game is 0.

2. ***We also tried the following changes to the program***

    a. *Use 2 instead of 4 as the depth.*

        By comparison, we found that the time required to run the program at a depth of 2 is much less than that at a depth of 4, and in our experiments, the winning rate of the program does not seem to reduce much. However, because the tutor mentioned in the discussion board that he hopes us to predict two actions for each player, we gave up depth 2 and continued to optimize the pruning method with depth 4.

    b. *Changed the evaluation function to make our strategy more aggressive.*

        We tried to make our strategy more aggressive by changing the evaluation function, but in the experiment, we found that this strategy has a lower winning rate, so we finally abandoned this change.

Other aspects:

1. ***We have designed many pruning methods to improve the efficiency and running time of the algorithm.***

Since only using minimax plus the alpha-beta pruning is not enough to reach a good efficiency, we decided to cut off most of the branch factors. Before cutting, even at the start of the game, there are already 5(positions)*3(symbols) throw actions. After a few times of actions, the branch factors increase with the token number. Besides those throw actions, each on-board token has 6 slide actions and some of them even have swing actions able to choose. Moreover, the Minimax algorithm requires us to simulate the opponent's actions. This explosive increasement exacerbates our time cost. We decided to cut off some actions which won't affect our ai's final output too much.

We remove three major part of actions:

a. Most throw actions. At the start of the game, we will randomly choose the position and type to throw on the first row of the board. Since we found if more tokens you keep on hand, it is more likely that you are able to beat the opponent, After the start, before player's total onboard token numbers over 4, if the opponent's one of type of token's number (e.g. symbol "r")exceed our token's number which the type can defeat the opponent's tokens (e.g. symbol"p"), we will only remain this type (e.g. symbol"p") of throw actions for our minimax algorithm search as branch factor. After our onboard token numbers over 4, the logic is almost the same as the previous one. The only difference is that if the opponent's one particular type tokens number (e.g. symbol "r") is more than 2 of the player's "greater" tokens' number (e.g. symbol "p"), it will find out appropriate throw actions for our search. In the meantime, we also restrict throw position. After the type is narrowed down to one, we find the position which is the closest to the opponent's token which our selected type can defeat. Then we give the action in which symbol and position are selected to the search algorithm as a branch factor.
b. Some tokens' all slide and swing actions. We think some tokens are not "urgent" as others. For example, some tokens are far from the opponent's tokens, or some tokens don't have opponents they can defeat. We will not let these tokens' actions be branch factors.
c. Normally, every token can have 6 slide options normally. However, now we randomly choose 3 slide options + 1 slide option toward the opponent's token which it can defeat.

Supporting work:

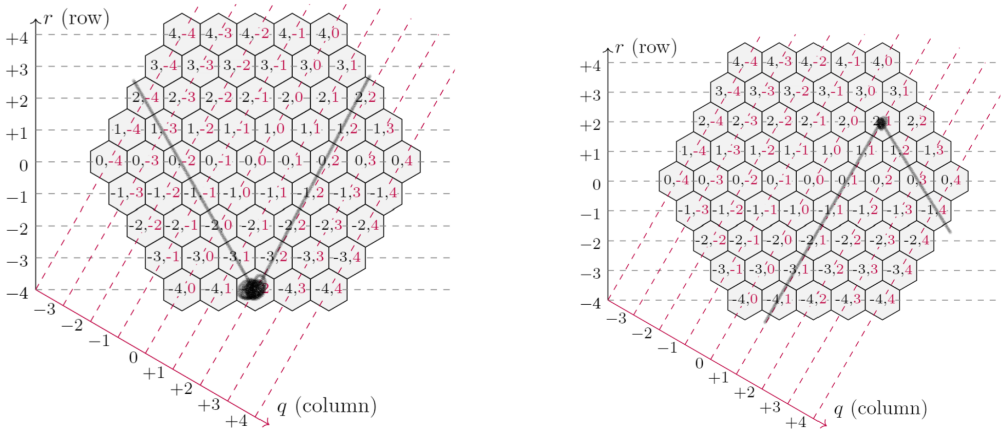
1. ***We designed a Gamestate class to help us view the entire board in the game.***

We designed a Gamestate class and instantiated an object through the game's _init_method. This gamestate object will store all the game information we should see in the game, including the current position of tokens, the number of throws made by us or the other party, etc.

There is an update method in the Gamestate class, whose function is not only to update the board based on the two incoming actions but also based on the game logic. When the update method of the game is called at the end of each turn, the update method of Gamestate will be called. It will update the board according to the game logic, find and delete invalid tokens to ensure that the board we see is exactly the same as the actual board.

2. ***We designed a distance calculation function to help us to evaluate the utility.***

The distance in this game is the number of steps from a hex to another hex.



Special v: Take origin hex as start point, draw two slash lines to up or down direction to form “v”. If the target hex is in this “v”, the distance is two hexs’ row difference.

If the target hex is higher than the origin hex, on v’s left is row difference +absolute value of the difference of two hexs’ sum of row and column index, on v’s right is row difference +absolute value of the difference of two hexs’ column index.
If the target hex is lower than the origin hex, on v’s left is row difference + absolute value of the difference of two hexs’ column index, on v’s right is row difference + absolute value of the difference of two hexs’ column index.