**Report for project othello**

**Name** Huaiwu Li

**ID** 11911425

1. **Preliminaries**

Project Description:

Othello is an extremely interesting game. Young people can play Othello to grow their wisdom and exercise their thinking skills, or to play with friends to spend time and have fun, while old people can play more chess to prevent Alzheimer's disease, and also to get unlimited happiness. This project is about Othello AI project, and it is full of fun to write the program and play chess.

My goal in this course project was to build an AI that tries its best to take fewer discs in Othello by using a search algorithm and an evaluation function. I used the alpha beta algorithm and an evaluation function that takes multiple factors into account and ended up with good results. In developing my AI program, I used the VScode editor and used the python language, which is version 3.6.13.

Problem Applications:

When you play on a gaming chess platform, do you wish you could play against stronger opponents to train yourself, help human players improve their ability or pass the time? YES, this program could do that! It is the worst AI ever could make you so confident about yourself even if you are a rookie in othello, it can make you win!

Besides the application of this course project is very broad, as the alphabeta algorithm and evaluation function can be applied not only to the construction of Othello ai and Othello games, but also to the construction of other games and to the solution of real-life problems such as path planning, such as finding locally optimal solutions.

1. **Methodology**

**Notation：**

*Evaluate()*:The evaluation function that evaluate the score of current state.

*Flip()*： Flip function is used to get the result chessboard after putting another chess on chessboard.

*Get\_candidate\_list()*: The function is used to get the corresponding candidate list for black or white.

*Isterminate()*: The function is used to test if the state is the final state to execute a evaluate function.

**Data Structure：**

*Candidate\_list*: a list of the most valuable positions which have found.

*Blackdic*: a dictionary stores all candidate\_list for black in current state of chessboard, and all solutions in this dictionary are calculated before, the key of the dictionary is current chessboard matrix and the value of the dictionary is a list.

*Whitedic*: Similar to Blackdic, a dictionary stores all candidate\_list for white in current state of chessboard, and all solutions in this dictionary are calculated before, the key of the dictionary is current chessboard matrix and the value of the dictionary is a list.

*Wmatrix*: a 2-dimensional matrix stores the importance value of each position on the chessboard, This matrix is used to calculate a simple weight value for the board position. The weight value is calculated by multiplying this weight matrix by the board matrix and then multiplying it by the color of the current ruling square, which gives a simple valuation of the board

*Wline*：This is a balanced list of weighting factors, which is used to balance the proportion of the total evaluation scores of different factors at different stages of the game, because different factors have different importance at different stages of the game, and therefore such a weighting list is needed to balance

**Model Design:**

1. **Problem Formulate**:

In this project I had many problems, some of them are critical, some are not, but all of them are directly or indirectly related to my final thinking and implementation process of the AI model, I will elaborate on all the problems I encountered and list the way I thought about them and their solutions.

The first category is the construction of the evaluation function. These problems revolve around judging the situation of Othello, in what situation this position is an advantageous position, and in what situation this position is an inferior position, and how to put the opponent in an inferior position, so that the opponent takes a bad position. Or even to give up some short term advantage for a long term, late advantage to get the final victory. Around these reflections, I generated many questions, which I tentatively grouped into the first category of questions about the evaluation function.

The second category of questions is about the choice of algorithm and the implementation of the algorithm, this type of problem is mainly to consider how to correctly implement the algorithm, as well as to make the algorithm simple, effective, and able to search more in the limited time and space resources, and this type of problem is often difficult to perceive, need to be careful and cautious to find the logical loopholes in the code, also need to repeatedly speculate to avoid falling into the trap.

The third type of problem is about code optimization, this type of problem does not seem to be as important as the above two problems, but if you ignore it, the result will leave you no mercy. On the contrary, such issues are of higher importance because the AI models we build are computed on a platform that relies on limited computing resources as well as storage resources. If we pay no attention to the existence of objective conditions, then even the best algorithms and evaluation functions are unlikely to achieve good results.

Regarding the first category of questions about evaluation functions, I ask these questions. How can a good evaluation function be constructed? What characteristics does a good evaluation function need to have? What are the aspects to consider in the specific implementation of the evaluation function? Should the evaluation function change with the phase of the game? How can the parameters be adjusted to achieve a good balance of the influences within the evaluation function? Regarding the second category of questions about algorithms, I also raised some questions. How to implement the alpha beta algorithm? How to find potential problems in the code, what are the main differences between the Alphabeta and minimax algorithms. And for the third category of optimization-related questions, I thought about how to optimize the code framework to make better use of time and space.

1. **Problem Solved**:

In solving the first type of problem, I took a combination of considerations and finally settled on using the four aspects of strength of action, potential strength of action and the difference in number of discs between the two sides as well as the board weighting score to obtain an evaluation score for the current position, and then multiplying these factors by the weighting to obtain the final score. In addition to this, I also filter for the specifics of the corners to get a different level of importance for each situation. And depending on the situation, I take different weights between these factors, so that I can also get different evaluation scores to suit the progress of the game.

For example, I use the flip() function to calculate the board situation after each move and pass it to the next level, and use the get\_candidate() function to get the location of the different color moves. There is also a judgment of the endgame and a judgment of the time limit, so that the return and evaluate() functions are executed when the time is close to the end or when no more discs can be played. Modularizing these functions into functions was very helpful for me to finally write code that worked well. Because if I can modularize the functions, I can test each function separately and structure the code more concisely and debug more easily, so that possible problems can be avoided and the code can be more portable and robust. The reason why the alphabeta algorithm is used instead of the minimax algorithm is that if the minimax algorithm is used, the number of search layers is less than the alphabeta algorithm in the limited computation time, and because of the many variations of Othello, there may be many changes in the position of the board if the opponent sees these If your opponent sees these changes and you don't find them because you don't have enough layers, then you have a natural disadvantage compared to your opponent.

In solving the third type of problem, I made a big optimization for both time and space. For the time optimization, I searched for n+1 layers after searching n layers, and the order of searching n+1 layers was cut to the opposite order of the candidate list, so that I could go one layer deeper according to the importance so that I can search in a deeper layer from high to low importance. And each search step will update the results in time, the results will be placed in the candidate list to avoid the loss of results for nothing. And because the search order is the opposite order of the candidate list, if there is a better score, it must be better than the previous results, and will not get worse results. And each search step will update the results, and keep searching until the end of time, so I will use up the time. For example, I will use the dictionary to record the computed candidate list of all the situations, so that in the deeper search I can use the previously recorded results instead of repeated calculations, so that the use of space to save time, the time and space resources are fully used to obtain the most computing power, so that relatively speaking there will be better results.

1. Main algorithm:

In the main algorithm section I will show the pseudo-code of the evaluation function and the pseudo-code of the alphabeta algorithm so that you have a clearer understanding and appreciation of my code implementation. Clear pseudocode and some supporting introductions are also necessary.

**Detail of algorithms:**

1. **Empirical Verification**

Dataset:

Performance:

Hyperparameters:

Experimental resutls:

Conclusion:

1. **References**