Minimax with alpha-beta pruning to play Othello

Siddhant Attri
Computer Science and Engineering
B18CSE052

1.Introduction

Othello is one of the classical games whose initial state is shown in Fig.1, and which can be solved by artificial intelligent methods that belong to search techniques in artificial intelligence. Many searching algorithms have been raised to solve this problem. In this project, some basic searching algorithms such as minimax search with alpha-beta pruning have been used to solve the problem.
In this classical game, two players take turns placing pieces-one player white and the other black-on an 8 × 8 grid board. Every move must capture one or more of the opponent's pieces. In order to capture, player1 should place a piece adjacent to one of player2's pieces, so that there is a straight line (horizontal, vertical, or diagonal) ending with player1's pieces and connected continuously by player2's pieces. Then the player2' pieces on the line will change to player1's. The final goal is to finish the game with as many pieces on the board as possible. A final state is shown in Fig.2, in which the black win obviously. In order to record the state, an 8×8 matrix will be built, and each element in this matrix indicates the state of the corresponding location on the chessboard. For example, it is 0 if the grid is empty, 1 if occupied by a white piece, and -1 if occupied by a black one. In programming, several steps should be involved: 1) checking moves, 2) making moves and 3) switching players.

Fig. 1. Initial state of Othello

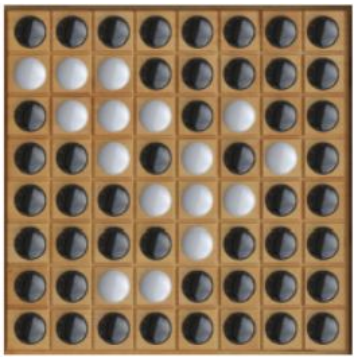


Fig. 2. Final state of Othello

2.Problem Definition

Generally, a board needs to be set up and initialized to the initial state as shown in Fig. 1. Then we need to check available moves for the current player according to Othello rules. Finally, the most important is how should the play move.
In my project, an 8 × 8 board is used. So, an 8 × 8 matrix is established that represents the initialized board as shown in Fig. 3. The "." are empty squares. 1 and 2 are black and white pieces respectively. In the long run, we should find a move that can achieve a higher reward, as well as trying to suppress the opponent's reward. So, a naive searching algorithm is to search for all possible moves for further steps and obtain a move with optimal reward.

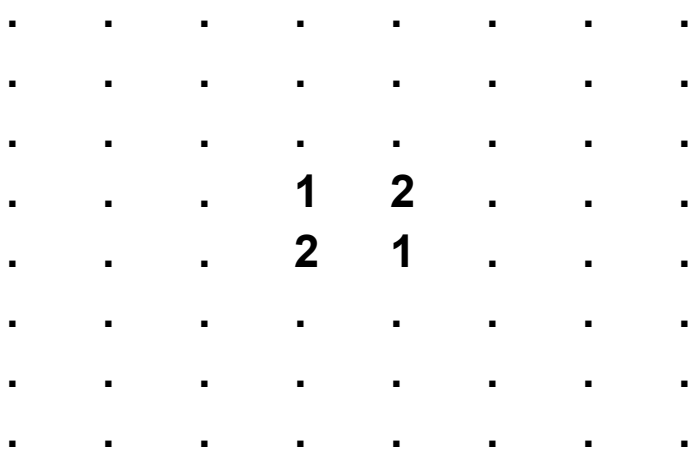


Fig. 3. Initialized Board

3.Background Survey

The following algorithms could be used to design an artificially intelligent opponent for the player.

3.1. Random

A random algorithm is the most simple one available. It just selects a random move from the available moves for the opponent.

3.2. Local Maximization

A more sophisticated strategy evaluates every available move according to a reward. This consists of getting a list of available moves, applying each one to a copy of the board, and selecting the move with the highest reward. Obviously, the local maximization algorithm is very short-sighted. Only one further step is considered.

3.3. Minimax Search Algorithm

A player who can consider the implications of a move several turns in advance would have a significant advantage. The minimax algorithm does just that. Fig. 4 shows the main idea of minimax searching. node A is player1, who will try all available moves and choose the one that obtains the best reward. nodes B, C, and D are the player2's turn after player1's moving. Player2 always wants to choose the move that makes player1 get the smallest reward. So, minimax searching is to alternatively maximize and minimize the rewards of child nodes, during which all available moves (nodes) will be evaluated. Obviously, this is a naive and time-consuming searching algorithm.

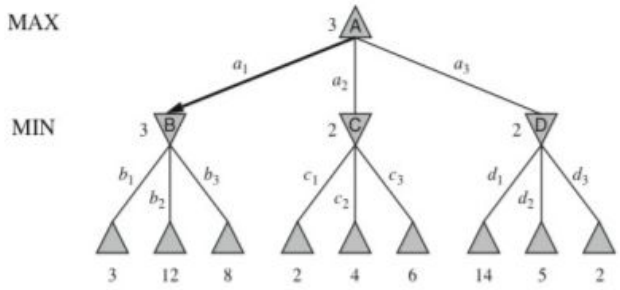


Fig. 4. Minimax Search Tree

4. Algorithm

I have used α-β pruning to design an AI opponent for the player. α-β pruning can be seen as an improvement of the minimax algorithm. It is more efficient since it seeks to decrease the number of nodes evaluated in its search tree. Specifically, it will stop evaluating a move when it has been proved to be worse than a previously examined move. Finally, the α-β pruning algorithm returns the same moves as minimax would, but prunes the branches that cannot affect the final decision. Therefore, α-β pruning costs much less time. Fig. 5 shows the basic idea of α-β pruning.

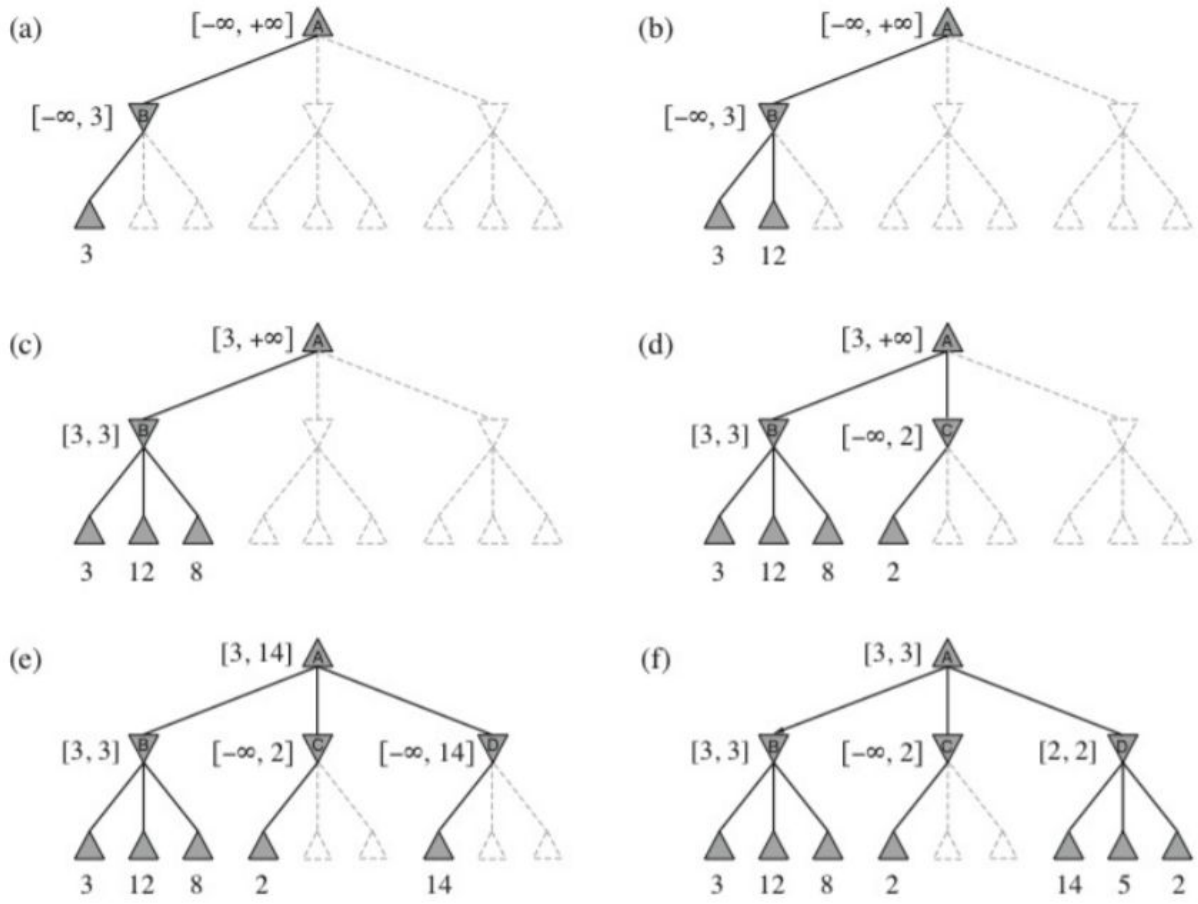


Fig.5. α-β pruning

In Fig. 5, the triangle is player1 who wants to maximize his reward, and the downward-point triangle is player2 who wants to minimize player1's reward. α is the best already explored reward for player1, and β is the best reward play1 can get under player2's suppression. Now, it's player1's turn (Fig. 5(a)). From node A, player1 chooses a move to node B, then it's player2's turn. Player2 chooses a move and makes player1 get a reward 3. Then player2 chooses another move that makes player1 get a reward 12 (Fig. 5(b)), etc. Player2 always wants to make player1 get the smallest reward, so the maximum reward play can get will be no more than 3 if player1 moves from node A to B (Fig. 6(c)). So the value of α can be updated to 3 since the current best reward player1 can get is 3. Similarly, player1 tries the next available move to node C. The first child node of C rewards 2 (Fig. 5(d)). Since player2 will choose the move that minimizes player1's reward, the reward player1 can get from node C will be never more than 2. Thus, node C will never be better than node B. According to α-β pruning, we will completely stop searching any other child nodes of node C. Then, we go on trying other moves (Fig. 5(e)). Again, when searching for the third child node of D, we

can confidently prune node D. The evaluation function that was used to calculate the goodness or the reward of the available states was a sum of the following values:

1. The piece difference i.e the difference between the white pieces and the black pieces when the move will be made. This component of the utility function captures the difference in coins between the max player and the min player.
2. Corner Captures i.e the number of corners that will be available with the player if they make that particular move. Corners hold special importance because once captured, they cannot be flanked by the opponent. They also allow a player to build coins around them and provide stability to the player's coins.
3. Close to Corner Pieces i.e the number of pieces on the boundary of the board owned by each player.
4. Stability: The stability measure of a coin is a quantitative representation of how vulnerable it is to being flanked. Coins can be classified as belonging to one of three categories: (i) stable, (ii) semi-stable, and (iii) unstable.

   Stable coins are coins which cannot be flanked at any point of time in the game from the given state. Unstable coins are those that could be flanked in the very next move. Semi-stable coins are those that could potentially be flanked at some point in the future, but they do not face the danger of being flanked immediately in the next move. Corners are always stable in nature, and by building upon corners, more coins become stable in the region.

   Weights are associated with each of the three categories and summed to give rise to a final stability value for the player. Typical weights could be 1 for stable coins, -1 for unstable coins, and 0 for semi-stable coins.

5. Mobility: It attempts to capture the relative difference between the number of possible moves for the max and the min players, with the intent of restricting the opponent's mobility and increasing one's own mobility.

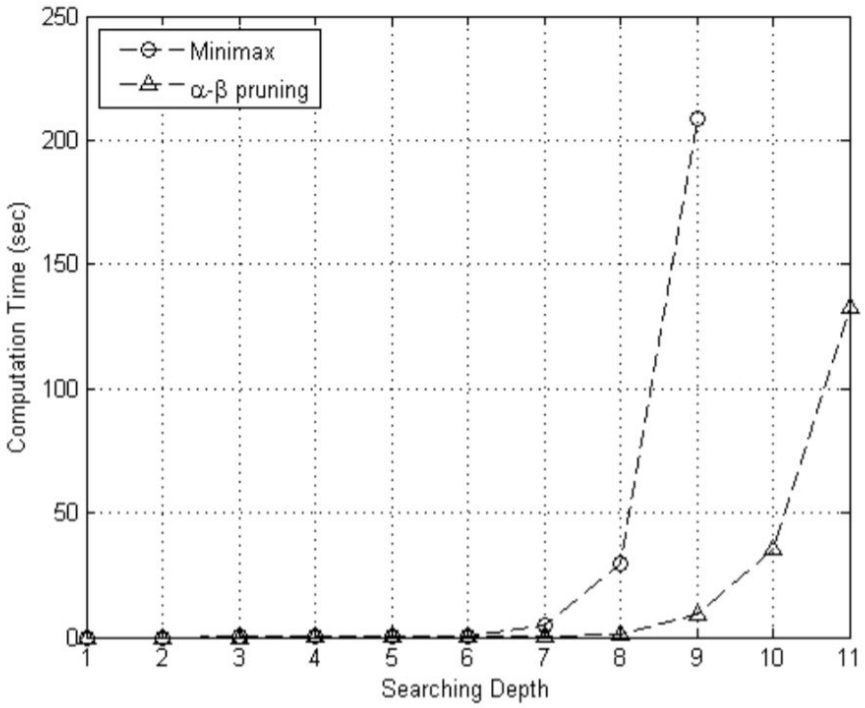
For a large search tree such as that of chess, even α-β pruning won't be sufficient enough to reduce the size of the search tree. And it is not practically possible to search such a large tree. For such situations, we use the expectimax search algorithm which is not guaranteed to find an optimal solution but would be practical for such large problems. The expectimax algorithm is a variation of the minimax algorithm, for use in AI systems that play two-player zero-sum games, such as backgammon, in which the outcome depends on a combination of the player's skill and chance elements such as dice rolls. In addition to "min" and "max" nodes of the traditional minimax tree, this

variant has "chance" nodes, which take the expected value of a random event occurring. In game theory terms, an expectimax tree is the game tree of an extensive-form game of perfect, but incomplete information.

5. Algorithm Complexity

Random and local maximization only consider the current state, so they search fast and have low time complexity. But they are not our analyzing points. The time complexity of minimax is $O(b^d)$, where b is the branching factor and d is searching depth. A bigger board will lead to a larger b. Usually, b can be seen as a constant. So the computation complexity of minimax will increase exponentially. For α-β pruning, the time complexity will drop to $O(b^{d/2})$. Fig. 6. Depicts a comparison of the time complexities of α-β pruning and minimax searching.

Fig. 6.



The space complexity of α-β pruning is $O(b^d)$ and for minimax search, this complexity remains the same.

6.Implementation

Pygame was used to construct a GUI for a human player to play against an AI opponent. Fig. 7 shows screenshots of the GUI in action.

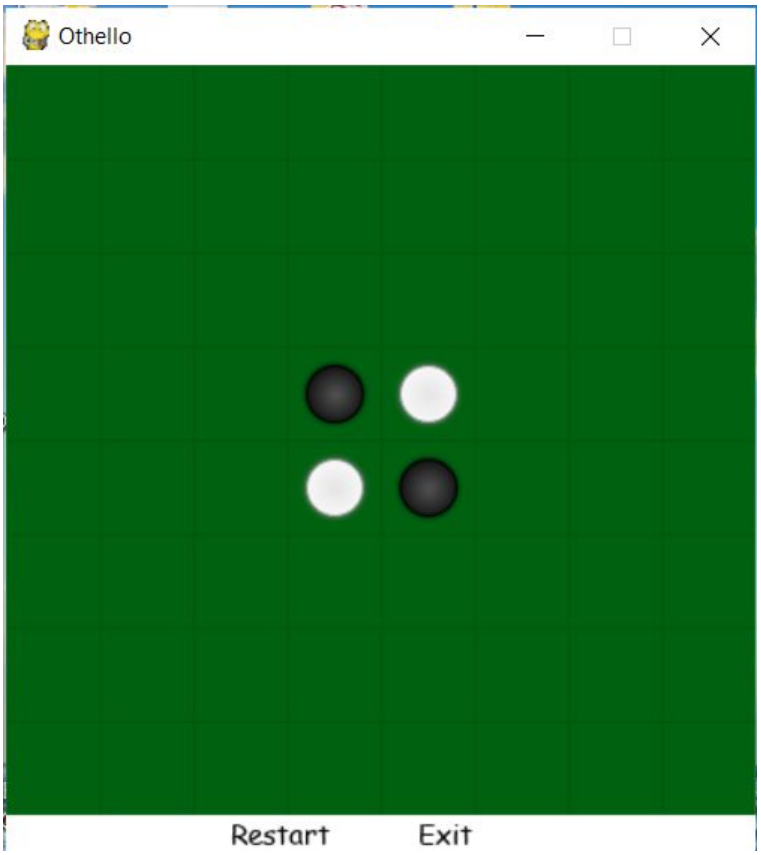


Fig. 7.1. Screenshot of the start state of the Othello implementation

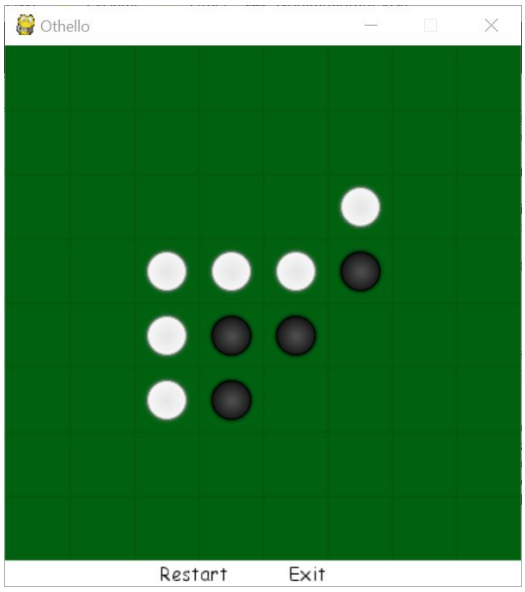


Fig. 7.2. Screenshot of some state of the Othello Implementation

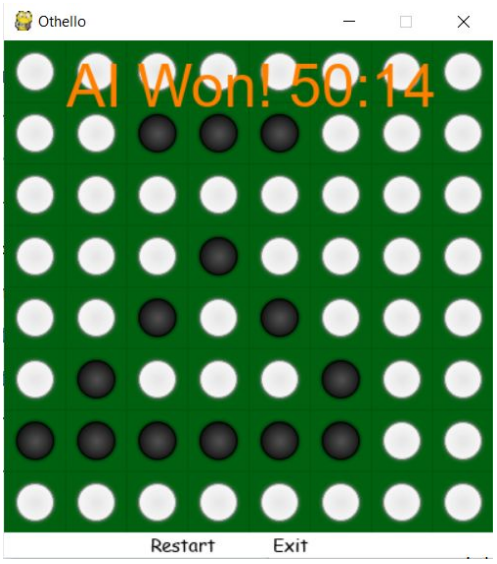


Fig. 7.3. Screenshot of the final state of the Othello implementation which was won by the AI

7. Conclusions

As we know, the most intelligent Othello algorithm can win the best human Othello players. That is a big success in AI study. In this report, one of these different algorithms were applied to Othello. For future work, it is interesting to know whether the

default policy can be improved more to get more significant performance. For example, it may be a good choice to use Othello's knowledge to handcraft a more intelligent search policy than α-β pruning such as expectimax search which could use a complex evaluation function to estimate the rewards for each move. This could help in saving a lot of time by not actually searching a deep tree but by providing estimates of the values of the deeper nodes to the higher tree levels.
Given a hard time constraint such as a few minutes my implementation would not be able to solve the problem because the AI opponent needs some time to search the game tree and make the best possible move i.e the move which would minimize the human opponent's score. This problem was hard exactly for this reason. That is why alpha-beta pruning was applied to improve the time complexity up to some extent. But say the board is a bit bigger then, my solution will not be able to solve this problem in a short time and will take a lot of computational time, say an hour. In such a situation an expectimax search could be used to solve the problem.

8. References

1. Stackoverflow.com
2. https://www.youtube.com/watch?v=y7AKtWGOPAE
3. Wikipedia.org
4. https://kartikkukreja.wordpress.com/2013/03/30/heuristic-function-for-reversiothello/