Implementing Minimax Search with Alpha-Beta Pruning in Blokus Duo

Timothy Niels Ruslim - 10123053
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: timothyniels@gmail.com , 10123053@mahasiswa.itb.ac.id

***Abstract*— This paper presents the implementation of a game-playing algorithm for *Blokus Duo* using the minimax search algorithm enhanced with alpha-beta pruning and transposition tables. *Blokus Duo*, a two-player strategy board game, offers a rich playground for adversarial search due to its special balance of difficulty. To evaluate game states, three heuristics are developed: score difference, mobility (corner availability), and centralization. These heuristics are analyzed independently, but also in combination through a weighted sum. The results show that while individual heuristics capture different strategic priorities at various game stages, the combined heuristic yields a robust intuitive human-like performance. Furthermore, optimization techniques such as careful move ordering and memoization can significantly reduce computation time. This work demonstrates how classical search techniques can be effectively applied in modern board games.**

Keywords—Blokus Duo; minimax algorithm; alpha-beta pruning; game trees; heuristic

I. Introduction

Blokus Duo is a two-player strategy-based board game now published by Mattel which serves as a simpler variant of the original *Blokus* game designed by Bernard Tavitian [2]. It is played on a 14 × 14 grid, using one white and one white set of 21 polyomino pieces each. Players alternate playing their pieces so that it touches at least one other piece of the same color but only at the corners. The first placement of a piece is dictated to be near the center of the grid. The game ends when no more valid moves, that is, piece placements, are possible, in which case the winner is determined to be the player with the least number of unplaced squares remaining [1].

Fig. 1. Blokus Duo

Due to its apparent simplicity yet complex strategies, striking a comfortable balance between tic-tac-toe and chess, *Blokus Duo* is a perfect strategy game for the amateur enthusiasts. In fact, it won the 2003 Mensa Select award and the 2004 Teacher's Choice Award [3]. With this, *Blokus Duo* becomes a particularly suitable sandbox for game strategy exploration. In particular, not being too difficult nor easy, it becomes the ideal domain for applying algorithmic decision-making, as this paper aims to explore.

More specifically, this paper aims to develop a game-playing algorithm for *Blokus Duo* to find optimal moves using the minimax search algorithm. This popular decision-making algorithm, that has been applied to numerous two-player strategy games like tic-tac-toe or chess, provides an amazing baseline for a *Blokus Duo* artificial intelligence. Moreover, optimizing techniques such as alpha-beta pruning and memoization will be further explored.

II. Theory

A. Blokus Duo Rules

The *Blokus Duo* game consists of a 14 × 14 grid board containing 196 squares. There are two sets of pieces for each player: one white and one black. Each set contains 21 polyomino pieces which includes 1 monomino, 1 domino, 2 tromino, 5 tetromino, and 12 pentomino.

The object of the game is to fit as much of the 21 pieces onto the board as possible under certain conditions. First, each player must place their first piece over the starting points, which are located near the center of the grid. Next, the play continues as each player lays down on piece at a time. Here, a new piece must be placed at least on other piece of the same color, but only at the corners. Thus, no flat edges of two pieces of the same color can touch. When a player is unable to place any of their remaining pieces, they must pass their turn. Finally, the game ends when both players are unable lay down any more pieces.

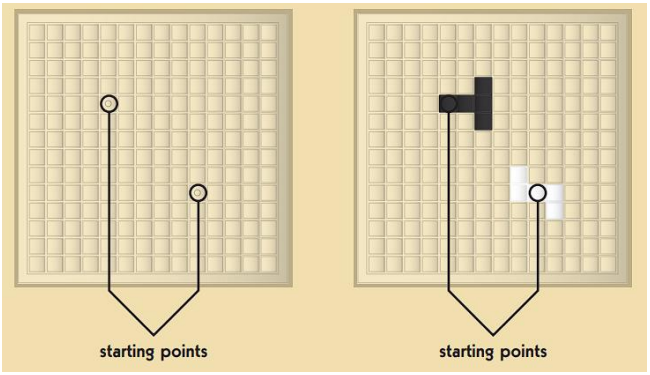


Fig. 2. Starting Points

To determine the winner, we assign a score to each player based on the remaining unplaced pieces. For each unplaced unit square in the remaining pieces, a player gets -1 point. In practice, however, one can also simply count the amount of unit squares that has been placed on the board, which will prove to be more convenient in translating this game to its digital version so that an algorithm can be applied. Then, if a player places all their pieces on the board, they earn an extra 15 points. If the last piece to be placed on the board is the smallest piece, the monomino, then the player also gets an extra 5 points [1].

B. Minimax Algorithm

A popular search algorithm often applied for decision-making in two-player zero-sum games is the minimax algorithm. The central idea of this algorithm is to simulate all possible future moves of both players in a dynamic game tree and evaluate the outcomes assuming each player plays optimally. Thus, each node of the tree represents a possible state of the game whilst each edge represents a possible move.

To determine the optimal move, as with other branch and bound algorithms, the minimax algorithm assigns a cost or weight to each node in the game tree. Its value is determined by a utility function, which assesses in which player's favor is a game state likely for. Sometimes, the utility function is trivially binary (win or lose) or is at least simple enough that it follows directly from the game scoring rules. For most games, however, the utility function must employ a certain heuristic to evaluate the profitability of a game state. Thus, there can be liberty and creativity in developing a minmax algorithm.

Now, unlike most branch and bound algorithms, the minimax algorithm optimizes the utility values of each node in a rather unique way. It simulates the existence of two actors: the maximizer and the minimizer. When building the tree at a certain depth, the algorithm assumes one of these two roles. As a maximizer, it wishes to maximize the utility value by choosing moves that increase the weight, which simulates the first-person trying to play the optimal move. On the other hand, the minimizer will minimize the utility value by playing the optimal move as an opponent [5].

In practice, the minimax algorithm generates by depth-limited depth-first search and is thus recursively implemented. The depth limit is often called the horizon. Once the algorithm reaches the terminal leaf node, it then evaluates the utility of those states. From those terminal nodes, minimax would then propagate the cost upwards, where for each level, if it is the maximizing player's turn, it would take the maximum value of the children nodes, whilst if it is the minimizing player's turn, it would take the minimum value of the children nodes. The following is the pseudocode for this procedure.

```
function minimax(node, depth, maximizingPlayer):
  if depth == 0 or isTerminal(node):
      return evaluate(node)

  if maximizingPlayer:
      maxEval = -∞
      for child in children(node):
          eval = minimax(child, depth - 1, false)
          maxEval = max(maxEval, eval)
      return maxEval
  else:
      minEval = +∞
      for child in children(node):
          eval = minimax(child, depth - 1, true)
          minEval = min(minEval, eval)
      return minEval
```

Fig. 3. Minimax Algorithm Pseudocode

In more mathematical terms, the min-max utility value of each game state can be calculated as follows. The minimax algorithm evaluates

$$\mathrm{Max}(s) = \max_{a \in A(s)} \mathrm{Min}\big(\mathrm{Result}(s,a)\big),$$

during the maximizer's turn and evaluates

$$\mathrm{Min}(s) = \min_{a \in A(s)} \mathrm{Max}\big(\mathrm{Result}(s,a)\big),$$

during the minimizer's turn. Here, $\mathrm{Result}(s,a)$ denotes the resulting state after applying a move a to a state s [4].

C. Alpha-Beta Pruning

For even modestly non-trivial games, the game search tree can be very large. In fact, it grows exponentially with an exponential runtime complexity of $O(b^d)$, where b is the branching factor and d is the horizon. To deal with this, one can employ an optimization technique called alpha-beta pruning, which as its name suggests, prunes branches from the game tree that does not affect the final decision.

Consider a node s and its unknown optimal utility value in the game tree. If there already exists a better choice t further up the tree from s, then the minimax algorithm would be inefficient to expand node s and evaluate its utility. Hence, one can prune the sub-tree of s. This is the core principle of the alpha-beta pruning optimization.

To do this, the algorithm maintains two values: α and β. Here, α represents the best possible score for the maximizing player so far, while β represents the best possible score for the minimizing player so far. When propagating the utility values up the tree from the terminal node, the values for α and β are updated accordingly. Then, before deciding to expand a certain node, the minimax algorithm can check first whether α and β is already a more optimal value so as not to expand the node. If

at any point $\alpha \geq \beta$, then the algorithm prunes the remaining children, which can save a lot of computation power. The following is the pseudocode modified to implement alpha-beta pruning [6].

```
function alphabeta(node, depth, α, β, maximizingPlayer):
  if depth == 0 or isTerminal(node):
      return evaluate(node)

  if maximizingPlayer:
      maxEval = -∞
      for child in children(node):
          eval = alphabeta(child, depth - 1, α, β, false)
          maxEval = max(maxEval, eval)
          α = max(α, eval)
          if β <= α:
              break
      return maxEval
  else:
      minEval = +∞
      for child in children(node):
          eval = alphabeta(child, depth - 1, α, β, true)
          minEval = min(minEval, eval)
          β = min(β, eval)
          if β <= α:
              break
      return minEval
```

Fig. 4. Alpha-Beta Pruning Algorithm Pseudocode

The effectiveness of alpha-beta pruning depends on the order of node searching. In the worst case, where the nodes are explored in the ordered of worst score, then there will be no pruning. In the best case, the nodes are explored in the order of best score, meaning all other children will be pruned. This yields an average runtime complexity of $O\left(b^{d/2}\right)$, which though still exponential, is still a massive improvement for large trees [7].

III. Implementation

To develop a minimax algorithm for *Blokus Duo*, the board game is translated into a Next.js web application using typescript. This is chosen to make the interaction more user-friendly and accessible online. Due to research limitations, the minimax algorithm will also be implemented using typescript, despite slower performance, to make things quick and easy.

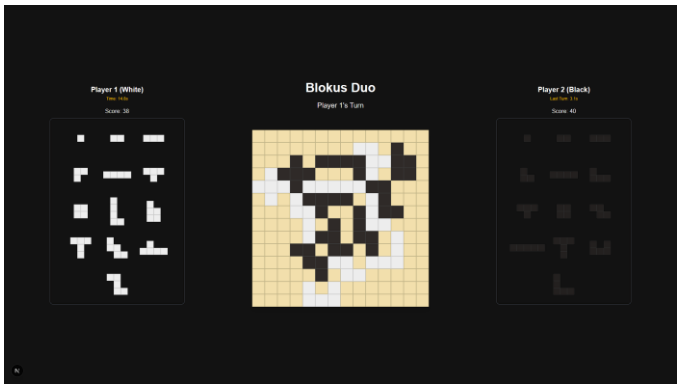


Fig. 5. *Blokus Duo* Application Interface

The interface developed can be seen in the figure above. The pieces are in trays, and users can simply drag and drop the pieces from their corresponding trays. One can rotate pieces by left clicking their pieces and flip them by right clicking. The code for the UI/UX can be seen in the GitHub repository attached.

A. Heuristics

First, some heuristics are determined to calculate the utility value (U) of a state, that is, the cost of a node in the game tree.

1. *Score Difference Heuristic*. This heuristic is simply the difference between the score (S), as dictated by the *Blokus Duo* rules, of the maximizing player and the minimizing player.

   $$U_1 = S_{\max} - S_{\min}$$

   Note that in the implementation of *Blokus Duo* for this project, the score counts the number of placed squares in the board instead of unplaced squares in the remaining pieces (which are don't actually affect the difference).

2. *Mobility Heuristic*. This heuristic counts the difference between the number of valid corner positions between the two different colors.

   $$U_2 = M_{max} - M_{min}$$

   The motivation for this heuristic is that a player would ideally want more corners available to them than the opponent, as this allows more opportunities to place pieces for the player, and less opportunities for the opponent to place pieces.

3. *Centralization Heuristic*. This heuristic is common in many two-player strategy turn-based games like chess. Most of the time, placing pieces closer to the center of the board is more profitable, as taking central territory can also increase mobility and is generally a more intuitive move. So, for any cell (i, j) containing the player's piece, we can calculate the following.

   $$\mathrm{C_p} = \sum_{(i,j)\in P} \left(\max\left(C_i, C_j\right) - \max\left(|i - C_i|, |j - C_j|\right)\right)$$

   This formula basically calculates the total difference between the Chebyshev distance of the squares from the center $\left(C_i, C_j\right)$ and the center coordinates.

   $$U_3 = C_{max} - C_{min}$$

4. *Combined Heuristic*. Finally, one can combine the three heuristics developed above into one single heuristic. To make it customizable, one can consider the weighted sum of the three scores.

   $$U = \sum_{i=1}^{3} w_i U_i$$

   This paper will arbitrarily choose $w_1 = 1$, $w_2 = 0.9$, and $w_3 = 0.5$ as the weights. These values are obtained simply by playing against the bot and tuning the results accordingly.

B. *Minimax*

The minimax algorithm that is implemented in this paper follows the pseudocode provided in theory, but with some minor adjustments. Below is the pseudocode for the minimax algorithm.

```
function minimax(
 state: GameState,
 depth: number,
 alpha: number,
 beta: number,
 isMaximizingPlayer: boolean,
 aiPlayer: 1 | 2,
 transpositionTable: TranspositionTable
): number {
 const boardKey = getBoardKey(state.board);
 const tableEntry = transpositionTable.get(boardKey);
 if (tableEntry && tableEntry.depth >= depth) {
    return tableEntry.score;
 }

 // Base case 1: reached max depth
 if (depth === 0 || state.gameOver) {
    return evaluate(state, aiPlayer);
 }

 const possibleMoves = generateMoves(state);

 // Base case 2: no more valid moves
 if (possibleMoves.length === 0) {
    return evaluate(state, aiPlayer);
 }

 const sortedMoves = possibleMoves.slice().sort((a,
b) => {
    const pieceASize =
a.piece.baseShape.reduce((total, row) => {
       const rowSum = row.reduce<number>((sum, cell)
=> sum + cell, 0);
       return total + rowSum;
      }, 0);

    const pieceBSize =
b.piece.baseShape.reduce((total, row) => {
       const rowSum = row.reduce<number>((sum, cell)
=> sum + cell, 0);
       return total + rowSum;
      }, 0);

    return pieceBSize - pieceASize;
 });

 let bestEval;
 // Maximizing player: alpha
 if (isMaximizingPlayer) {
    bestEval = -Infinity;
    for (const move of sortedMoves) {
       const childState = applyMove(state, move);
       const evalScore = minimax(
          childState,
          depth - 1,
          alpha,
          beta,
          false,
          aiPlayer,
          transpositionTable
       );
       bestEval = Math.max(bestEval, evalScore);
       alpha = Math.max(alpha, evalScore);
       // Prune
       if (beta <= alpha) {
          break;
       }
    }
 }

 // Minimizing player: beta
 else {
    bestEval = Infinity;
    for (const move of sortedMoves) {
       const childState = applyMove(state, move);
       const evalScore = minimax(
          childState,
          depth - 1,
          alpha,
          beta,
          true,
          aiPlayer,
          transpositionTable
       );
       bestEval = Math.min(bestEval, evalScore);
       beta = Math.min(beta, evalScore);
       // Prune
       if (beta <= alpha) {
          break;
       }
    }
 }

 transpositionTable.set(boardKey, {score: bestEval,
depth: depth});
 return bestEval;
}
```

Fig. 6. *Minimax Algorithm for Blokus Duo*

Notice that in this case, a terminal node is reached when either the horizon depth is reached, the game is over, or there are no more pieces left to place. Then, the recursive implementation hints the fact that the utility values calculated using the heuristics that has been discussed will be propagated up the tree. Then, with the use of alpha-beta pruning, the algorithm can smartly skip the traversal of certain nodes whose utility values are not optimal compared to those already searched.

```
export async function findBestMove(state: GameState,
depth: number): Promise<Move | null> {
   const aiPlayer = state.currentPlayer;
   let possibleMoves = generateMoves(state);

   if (possibleMoves.length === 0) {
      return null;
   }

   const transpositionTable: TranspositionTable = new
Map();

   const sortedMoves = possibleMoves.slice().sort((a,
b) => {
     const pieceASize =
a.piece.baseShape.reduce((total, row) => {
       const rowSum = row.reduce<number>((sum, cell)
=> sum + cell, 0);
       return total + rowSum;
      }, 0);
```

```
     const pieceBSize =
b.piece.baseShape.reduce((total, row) => {
       const rowSum = row.reduce<number>((sum, cell)
=> sum + cell, 0);
       return total + rowSum;
      }, 0);

     return pieceBSize - pieceASize;
  });

  let bestMove: Move | null = sortedMoves[0];
  let maxEval = -Infinity;

  // Iterate through every move
  for (const move of sortedMoves) {
     const childState = applyMove(state, move);
     const evalScore = minimax(
        childState,
        depth - 1,
        -Infinity,
        Infinity,
        false,
        aiPlayer,
        transpositionTable
     );

     // Found better move
     if (evalScore > maxEval) {
        maxEval = evalScore;
        bestMove = move;
     }

     await new Promise((resolve) =>
setTimeout(resolve, 0));
   }

   return bestMove;
}
```

Fig. 7. *Finding Best Move in Blokus Duo*

To find the best move, however, one must iterate through the children of the root state and then pick the state with the highest utility value, as the minmax algorithm only returns the weight, not the best move. This is what the `findBestMove` function in the figure above achieves.

C. Optimizations

Other than alpha-beta pruning, some other optimizations were implemented. First, recall that the order at which the nodes in the game tree is traversed affects the capabilities of alpha-beta pruning. A desired ordering is that the more optimal moves are traversed first. This problem on its own requires a priority heuristic. For this paper, the heuristic chosen is simply to order the moves by those whose pieces have more squares, as this is more likely to yield a greater material score.

Next, to further speed up the minmax search, transposition tables are implemented. They serve as cache memory to store game states that have been visited in the search earlier. The term "transposition" here is used to denote the arrival of the same position despite going through different sequence of moves [8]. When a unique state is traversed, the algorithm stores the position and its optimal value in a transposition table. Practically, this is a hash map.

IV. Results and Analysis

The final developed *Blokus Duo* minimax algorithm using alpha-beta pruning can be checked in the repository attached.

A. Comparing Heuristics

To test the performance of the minimax algorithm, the different heuristics (score, mobility, centralization, and combined) developed will be compared on different positions to see the different moves and times taken. For ease of use, a depth horizon of 2 will be used throughout the testing. The rationale behind this is also the performance issue that comes with creating the algorithm using JavaScript instead of a quickly compiled language.

Initial State. First, the algorithm will be tested on the initial board. Here, obviously it is white to move first. The chosen optimal moves based on each heuristic following the position in the figure above is summarized in the following table.

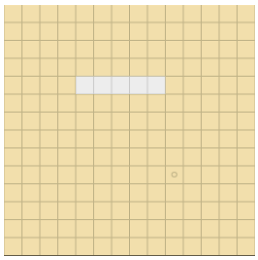
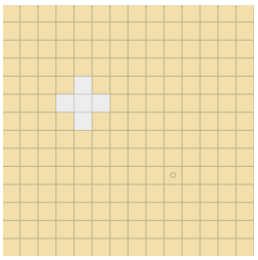
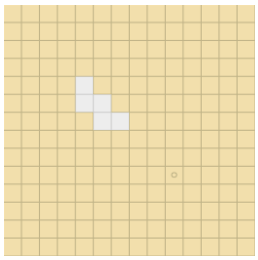
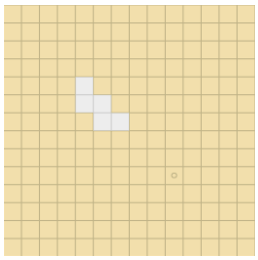
Heuristic	Move	Time
Score		3.6 s
Mobility		4.3 s
Centralization		3.0 s
Combination		4.3 s

Fig. 8. *Initial State Results*

Midgame State. Next, a midgame state as in the following figure is tested. It is white to move.

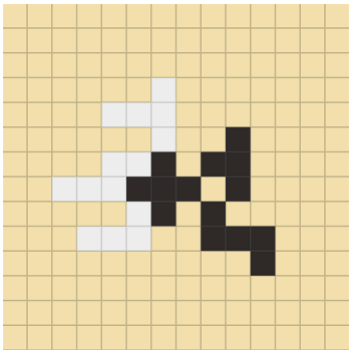


Fig. 9. *Sample Midgame State*

The chosen optimal moves based on each heuristic following the position in the figure above is summarized in the following table.

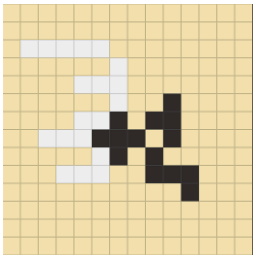
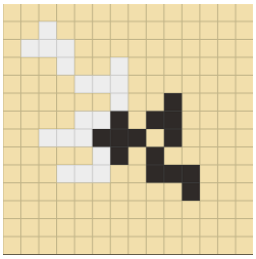
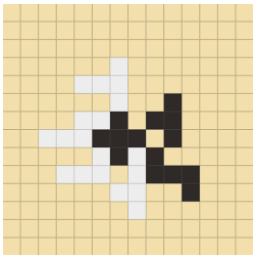
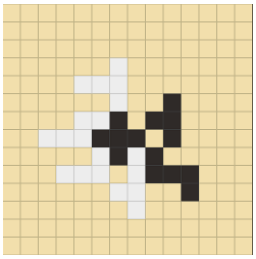
Heuristic	Move	Time
Score		6.8 s
Mobility		6.5 s
Centralization		6.4 s
Combination		6.9 s

Fig. 10. *Midgame State Results*

Endgame state. Finally, an endgame state as in the following figure is tested. It is black to move.

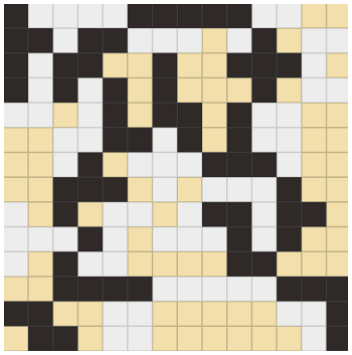


Fig. 11. *Sample Endgame State*

The chosen optimal moves based on each heuristic following the position in the figure above is summarized in the following table.

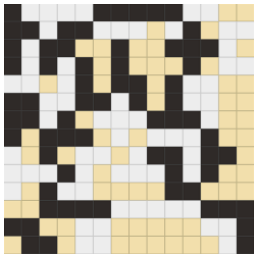
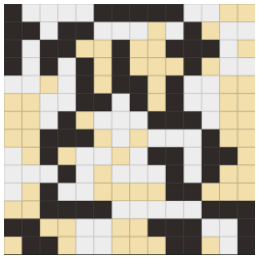
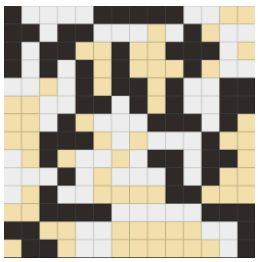
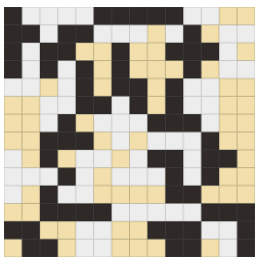
Heuristic	Move	Time
Score		0 s
Mobility		0 s
Centralization		0 s
Combination		0 s

Fig. 12. *Endgame State Results*

B. *Analysis*

Based on the empirical results above, each heuristic captures a different strategic part of the game, and their effectiveness depends too on the stage of the gameplay. However, computational time between heuristics do not vary much, for that depends largely on the nodes traversed, not the accumulated time for heuristic calculations.

In the initial state, the characteristic of each heuristic becomes most obvious. The score heuristic simply goes for the first pentomino in the move order, the mobility heuristic goes for the piece with the most corners, and the centralization heuristic goes for the piece that is most diagonal. From a human standpoint, certainly the score heuristic and centralization heuristic results in the most intuitive decision. This is most reflected in the combination heuristic, which after weighing in all three opinions, chooses the same “W5” pentomino move as the centralization heuristic.

In the midgame state, the score and mobility heuristic play quite unintuitive moves for a human. In their attempt to either maximize score differential or corner square potential, the algorithm yields decisions that place larger pieces in more open spaces. On the other hand, centralization and combination heuristics result in a more offensive attack, as it places pentominos near the center where black is.

In the endgame state, all four heuristics go for the “P5” pentomino, which makes sense as with the limited space left, going for the piece with the highest number of internal squares, as well as allow for more corner spaces after places, is only rational. The different placements of the “P5” piece, though, is more a reflection of the move ordering and little space left. Notice too that the algorithm is quick during endgames, further showing the weight of the exponential search runtime.

Overall, this shows that the combined weighted heuristic is the most robust across all stages of play. Even though the weights were tuned heuristically, they provide a balanced strategy that approximates rational human play. Moreover, the performance benefits of alpha-beta pruning and transposition tables are evident in keeping evaluation times manageable.

V. Conclusion

In this paper, a minimax algorithm using alpha-beta pruning and transposition tables was developed to determine the best moves in the two-player strategy-based game *Blokus Duo*. Three main heuristics were explored: a score heuristic, a mobility heuristic, and a centralization heuristic. Though in themselves each heuristic might yield rather bizarre or unintuitive moves, a weighted combination of them admissibly mimics intuitive human *Blokus Duo* players.

For future research, it is advised to further optimize the algorithm by using faster compiled languages such as C++ or Rust to run the minimax search. This way, deeper horizons can be used to create a smarter algorithm. One can also explore the principle variation search (Negascout) paired with better move orderings to yield more efficient algorithms.

Repository Link (Github)

https://github.com/timoruslim/blokus-duo

Acknowledgment

The writer would like to thank everyone who is involved and contributed to the writing of this paper. This gratitude extends to those who has given support and love throughout the course of this Algorithm Strategies course. A special thanks to Dr. Nur Ulfa Maulidevi, S.T, M.Sc. as the teaching professor of the Algorithm Strategies class 01, who has educated all the students and thus heavily contributed to the development of this research.

References

[1] Mattel, "Blokus Duo Instructions," 2017. [Online]. Available: https://www.buffalolib.org/sites/default/files/gaming-unplugged/inst/Blokus%20Duo%20Instructions.pdf

[2] J. Glenn and E. F. Larsen, UNBORED Games: Serious Fun for Everyone. New York, NY: Bloomsbury USA, 2014.

[3] "Winning Games," Mensa Mind Games. [Online]. Available: https://www.mensamindgames.com/about/winning-games/

[4] "Mini-Max Algorithm in Artificial Intelligence," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/artificial-intelligence/mini-max-algorithm-in-artificial-intelligence/

[5] E. Roberts, "The Minimax Algorithm," Stanford University, 2003. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/minimax.html

[6] E. Roberts, "Alpha-Beta Pruning," Stanford University, 2003. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/alphabeta.html

[7] "Alpha Beta Pruning in AI," MyGreatLearning. [Online]. Available: https://www.mygreatlearning.com/blog/alpha-beta-pruning-in-ai/

[8] T. A. Marsland, "The Anatomy of Chess Programs," University of Alberta. [Online]. Available: https://webdocs.cs.ualberta.ca/~tony/ICCA/anatomy.html

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Juni 2025

Timothy Niels Ruslim (10123053)