Sorting Algorithms

Project 3
Algorithm Design and Analysis

Danil Krassotov 282656

Introduction

The purpose of this report is to implement and analyze the effectiveness of an artificial intelligence algorithm in the context of a checkers game. As part of the project, a checkers game was implemented using the **Minimax** algorithm with *Alpha-Beta* pruning. Measurements were conducted for the algorithm to evaluate its performance. Based on the collected data, graphs will be generated and an analysis of the results will be presented, which will allow for drawing conclusions.

1 Basic Game Logic

The main logic of the checkers game was built based on several cooperating classes that represent the basic elements of the game and manage its course. These modules are responsible for the board state, piece and move definitions, as well as overall game control.

1.1 Representation of Game Elements

To model the game state, three basic structures were defined: ‘Position‘, ‘Piece‘ and ‘Move‘.

‘Position‘ Class

The ‘Position‘ structure is used to represent the coordinates of a field on the board. It consists of two components: ‘row‘ and ‘col‘ (column).

‘Piece‘ Class

The ‘Piece‘ class represents a single piece on the board. It stores information about its color (white or black) and type (piece or king).

‘Tile‘ Class

The ‘Tile‘ class represents a single field on the board. Each field can contain a piece or be empty. This class also stores the field coordinates (‘row‘, ‘col‘) and highlighting status (‘highlighted‘), used mainly by the visualization module.

‘Move‘ Class

The ‘Move‘ class encapsulates a single move in the game, containing information about the starting position (‘from‘), ending position (‘to‘) and a list of captured piece positions (‘captured‘). This is crucial for handling multiple captures in checkers.

‘Board‘ Class

The ‘Board‘ class is the central element of the game logic, responsible for maintaining the board state, initialization, move validation and their application. The board is represented as a two-dimensional array of ‘Tile‘ objects.

‘Game‘ Class

The ‘Game‘ class is responsible for the entire game course. It handles game initialization, player turn management and checking for game end conditions.

2 Artificial Intelligence (AI)

The artificial intelligence module is responsible for making move decisions for the computer. It uses a game tree search algorithm, supported by a board state evaluation function, to select the optimal move.

2.1 Minimax Algorithm with Alpha-Beta Pruning

The foundation of the AI operation is the **Minimax** algorithm with optimization in the form of **Alpha-Beta pruning**. Minimax is a recursive algorithm used for making optimal decisions. It works by searching the game tree, where nodes represent board states and edges represent possible moves. The goal of the maximizing player (AI) is to maximize the evaluation function value, while the minimizing player (opponent) aims to minimize this value.

Alpha-Beta pruning is an optimization technique that significantly reduces the number of nodes that must be visited in the search tree, without affecting the final result. It works by eliminating branches that certainly do not lead to an optimal solution.

- **Alpha** (for the maximizing player): Stores the best value found so far for the branch being examined by the maximizing player.
- **Beta** (for the minimizing player): Stores the best value found so far for the branch being examined by the minimizing player.

If during tree search Alpha becomes greater than or equal to Beta ($\alpha \geq \beta$), it means that the current branch is worse than the already found path and can be "cut off", i.e., further search can be terminated.

2.2 Evaluation Function

The ‘evaluateBoard()‘ function is a key element of the AI, as it assigns a numerical value to a given board state. The higher the value, the better the position for the AI player. The evaluation is based on several factors.

- **Piece value**: Pieces - 100, kings - 300;
- **Mobility**: Number of available moves. Greater mobility is beneficial.
- **Progress**: Rewards pieces that have moved further toward the opponent’s row.

- **Center control**: Prefers pieces located closer to the center of the board, which are strategically important.
- **Cowardice penalty**: Penalizes pieces located on the board edges, as they have limited mobility.
- **Last row bonus**: Rewards pieces that are in the last row (ready for promotion to kings or already as kings).

2.3 Computational Complexity

In the worst case, without cuts, the complexity is $O(b^d)$. Thanks to Alpha-Beta cuts, the average complexity is reduced to $O(b^{d/2})$, which significantly speeds up the algorithm and allows searching greater depths.

3 Measurement Analysis

To evaluate the performance of the implemented artificial intelligence (AI) algorithm, a series of measurements was conducted on the time needed to find the optimal move. Measurements were performed for different search depths, from 1 to 6. For each depth, 5 tests were conducted, and the results are presented as average, minimum and maximum execution time values.

3.1 Measurement Results

The following table presents the collected data regarding AI execution time in milliseconds (ms) depending on search depth.

Table 1: AI algorithm performance measurement results

Depth	Average (ms)	Min (ms)	Max (ms)	Tests
1	0.35	0.35	0.35	5
2	2.67	2.38	2.93	5
3	9.87	9.64	9.98	5
4	30.87	30.22	32.52	5
5	133.84	132.62	135.59	5
6	448.41	441.23	459.29	5

3.2 Graphs and Interpretation

The following graph shows the average execution time of the AI algorithm as a function of search depth. The Y-axis is on a logarithmic scale, which allows better observation of the exponential nature of time growth.

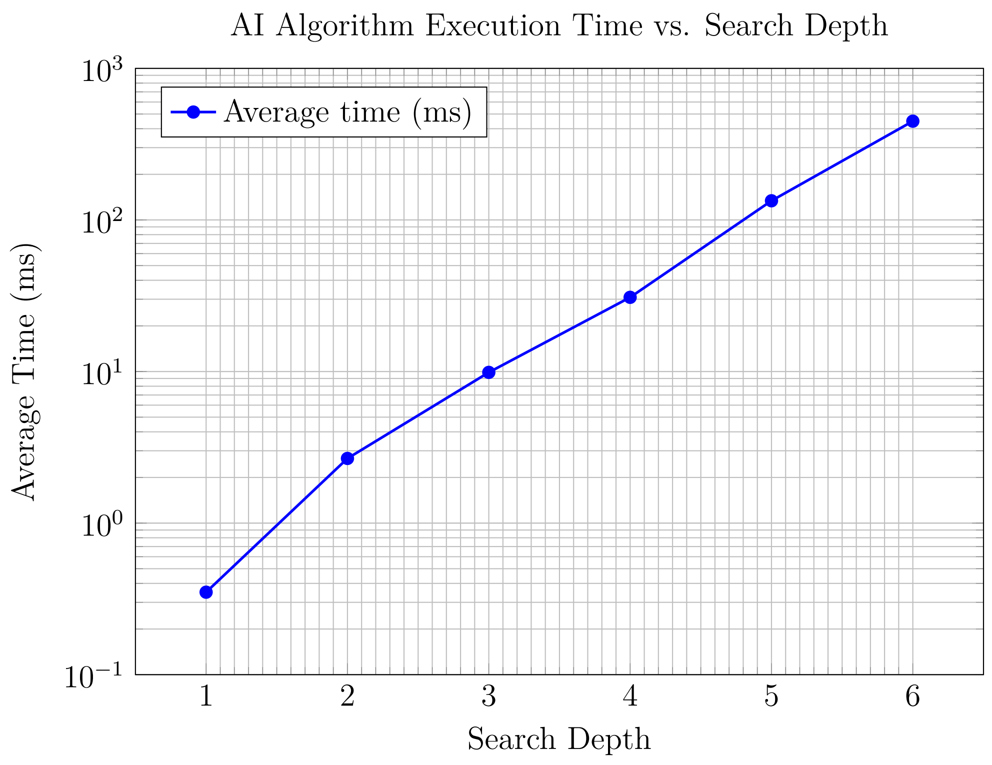

Figure 1: Graph of AI algorithm execution time depending on search depth (logarithmic scale on Y-axis)

Time Complexity Analysis

In accordance with theoretical expectations for the Minimax algorithm with Alpha-Beta cuts, execution time grows exponentially with increasing search depth. On the graph, despite using a logarithmic scale on the Y-axis, non-linear growth is clearly visible.

The presented time ratios between consecutive depths confirm this:

- Ratio of depth 2 time to depth 1 time: 7.69x
- Ratio of depth 3 time to depth 2 time: 3.70x
- Ratio of depth 4 time to depth 3 time: 3.13x
- Ratio of depth 5 time to depth 4 time: 4.34x
- Ratio of depth 6 time to depth 5 time: 3.35x

The theoretical complexity for Alpha-Beta is $O(b^{d/2})$, where b is the branching factor and d is the depth. The observed results are consistent with this model. For deeper searches (e.g., above 6), execution time quickly becomes unacceptable for real-time gameplay.

The measurements confirm that the AI algorithm, while effective in finding good moves, requires optimization to achieve greater search depth in reasonable time.

4 Conclusions

The key element of the project is the **Artificial Intelligence module**, utilizing the **Minimax algorithm with Alpha-Beta cuts**. This algorithm effectively searches the game tree, and the application of cuts reduces the number of analyzed board states, which is crucial for achieving reasonable response times. The evaluation function allows the AI to make rational and often optimal decisions.

Performance measurement analysis confirmed theoretical predictions regarding computational complexity. The AI algorithm execution time grows exponentially with increasing search depth, which is characteristic of game tree search problems. Although Alpha-Beta cuts significantly reduce this growth, already at depth 6 the times become significant (over 400 ms). This shows that to achieve greater AI power, further optimizations are necessary.

In summary, the project offers both a functional checkers game and a demonstration of AI algorithms. The implemented approach is scalable and I would gladly improve the project after the session.