# Project Report: Algorithms and Data Structures

Davide Luccioli
0001028403

A.A. 2021-2022
Alma Mater Studiorum – Università di Bologna

## Introduction

**Objectives:**
The project consists of creating a software player capable of optimally playing all possible configurations of an (M, N, K)-Game: a perfect information, zero-sum game where two players take turns marking a square on an MxN grid with their symbol (X or O) to achieve K consecutive marked squares (vertically, horizontally, or diagonally).

The player must implement the `MNKPlayer` interface, provided with the `mnkgame` package, which requires methods for player initialization and selecting the square to mark in the current turn.

**The Problem:**
Selecting the square to mark is central to the project, as the player must decide intelligently, within a time limit (set to 10 seconds), which move to make. Typically, zero-sum strategy games are represented as decision problems on trees: each game board state is seen as a node in a game tree, where the root represents the initial state of the board, and the leaves are the final states, with edges representing player moves. In an ideal scenario, this problem can be solved by exploring the entire tree using algorithms like MiniMax, which assigns a score to each node to identify the best move. However, exploring the entire game tree is often infeasible due to the large number of nodes, even with optimizations like Alpha-Beta Pruning.

Thus, the goal is to develop an algorithm that allows the player to:

- Explore as many nodes as possible within the time limit,
- Stop exploration before the time expires,

- Evaluate intermediate nodes using a heuristic function.

# Design Choices

**Evaluation Function:**
The player's move selection is based on the MiniMax algorithm optimized with Alpha-Beta Pruning with limited depth. Exploration of a game tree branch stops and the game state is evaluated when a leaf node is reached (the game ends) or the maximum depth is reached. For leaf nodes, a score of 1,000,000 is assigned for a win, -1,000,000 for a loss, and 0 for a draw. Wins at lower depths are considered better, and losses at lower depths are worse, factoring the distance from the maximum depth into the evaluations.

Intermediate nodes are evaluated based on a heuristic that identifies threats, defined as particular symbol alignments that bring the player closer to a win. Threats are categorized into three types:

1. **Open threats**: Consecutive symbols with both ends open.
2. **Semi-open threats**: Consecutive symbols with one end open or symbols separated by one empty square.
3. **Closed threats**: Consecutive symbols with both ends blocked.

For each cell on the board, adjacent cells on lines (up-right, right, down-right, and down) are checked to evaluate the game state. Scores are assigned based on the number of consecutive symbols in a line and the type of threat it forms. The evaluation of the game state is the sum of the scores of all threats on the board.

The scoring is inspired by the heuristic proposed by Abdoulaye-Houndji-Ezin-Aglin [1], where the most important threats are K-1 length (both open and semi-open), as they directly lead to a win. Semi-open threats of length K-2 and smaller threats are less relevant, so they are scored based on the number of occupied cells. Player threats receive positive scores, while opponent threats receive negative scores, weighted more heavily in favor of the opponent to prioritize blocking the opponent over improving the player's own position.

**Iterative Deepening:**
To maximize the explored depth, the player employs iterative deepening: the game tree is traversed multiple times from the root with increasing depth limits. With each iteration, a more accurate score for available moves is obtained. The algorithm stops when the maximum depth is reached or the

available time is about to expire, at which point the best move from the last completed iteration is returned.

Iterative deepening provides better control over the available time without additional computational costs. Despite repeated explorations, the computational cost is comparable to Alpha-Beta Pruning since most explored nodes are in the lower levels of the tree.

**Transposition Tables:**
It is common for the same game state to appear in different branches of the game tree, called transpositions. To avoid re-evaluating transpositions, the player uses a transposition table: a large hash table where explored game states are saved. This avoids redundant searches, as previously calculated scores are reused.

Occasionally, a score saved in the table cannot be used because it may be from a lower-depth search and thus less accurate. Moreover, the score from Alpha-Beta Pruning might be an upper or lower bound, not the exact score. In these cases, the table can still speed up recalculations by adjusting the alpha and beta values. Additionally, the best move found for a state in the last search is saved, and this move is always explored first in subsequent searches, increasing the chances of a cutoff.

**Zobrist Hashing:**
In the transposition table, game states are saved as keys generated using Zobrist hashing, which creates an MxNx2 table containing 64-bit pseudo-random numbers. For each cell in the game board, two numbers correspond to the symbols that can occupy it. Each time a cell is marked, the game state key is updated using an XOR between the current key and the number corresponding to the marked cell.

**Collision Handling:**
**Key Collisions:** Zobrist hashing can produce key collisions, where two different game states result in the same key. The use of 64-bit keys minimizes the likelihood of this occurring.

**Index Collisions:** To reduce the number of index collisions in the table, a prime number is used as the table size. Collisions are handled based on the depth of the explored node: an entry is replaced by another entry with the same index only if the new node's depth is greater or equal to the previous node's depth.

# Computational Cost

The algorithm's computation time is limited by the maximum time assigned during initialization, and the iterative deepening loop stops when the avail-

able time is about to expire. The player's performance depends on the computational cost of the `alphaBeta()` method. The lower the cost, the more cycles of iterative deepening can be completed, improving move evaluation.

A complete analysis of computational cost is complex, as it depends on the order in which moves are explored. In the best case, the algorithm always explores the best move at each level, yielding a cost of $O(\sqrt{n!})$, where $n$ is the number of available moves. In the worst case, no cutoffs occur, and the moves are explored from the worst to the best, resulting in a cost of $O(n!)$. However, using a transposition table along with iterative deepening improves the ordering of tree nodes, as the first move explored is always the best from the previous search. As a result, the average search cost approaches the optimal case.

# Possible Improvements

Several tree-based decision algorithms, such as MTD(f) or NegaScout, could offer better performance than the MiniMax algorithm used in this project. Additionally, further optimizations could be achieved by identifying symmetries between game states, as symmetric nodes result in the same evaluation. Recognizing these symmetries would allow the avoidance of evaluating nodes whose scores have already been calculated.

# Conclusions

Although still optimizable, the implemented player is able to compete in all tested instances of (M, N, K)-Game, making sufficiently intelligent moves for each game state. The techniques employed allow the exploration of a large number of nodes, resulting in fairly accurate evaluations of the game state and leading to good move choices in most situations.

# References

[1] Abdel-Hafiz Abdoulaye, Vinasetan Houndji, Eugène Ezin, and Gael Aglin. Cari2018 p265-275. 10 2018.