LombEnis: strategic optimization for connectx

1 Introduction to the computational problem

This project focuses on the implementation of an algorithm for the game **ConnectX**, a variant of the well-known game Connect 4. This project addresses an intriguing computational challenge, which consists in developing an optimal strategy in order to make the best move in a given game context, within a pre-established time.

The variability in grid size and number of pieces required to win makes ConnectX a problem that touches on several topics in algorithms and data structures. Our main goal in this project was to develop an algorithm capable of quickly evaluating the current game situation and selecting the optimal move to maximize the chances of victory or minimize possible defeat, depending on the context.

In this report we will take an in-depth look at the computational problem of ConnectX. We will discuss the specific challenges associated with the game, the approaches used to address them, and the results obtained through the implementation of the algorithm. An analysis of the **algorithmic strategies** employed to address this problem will be presented, including the data structures used to represent the algorithm's decision-making processes and to improve its execution speed. Finally we will approximately evaluate the **computational cost** of the algorithm in order to understand its efficiency.

2 Design choices adopted

In this section, we will examine the **design choices** made to create an accurate and efficient algorithm. Please note that the code underlying these choices has been carefully organized and commented to ensure readability and comprehensibility, here we will provide a more high-level view of the key choices, which have been divided into three distinct phases for a better understanding of the development process.

2.1 Choice of decision algorithm

The first phase of the project involved the selection of the **decision algorithm** to be adopted.

From the beginning, we opted to use the **minimax** algorithm, recognized as suitable for strategy games where the goal is to minimize losses and maximize the chances of winning.

However, given the variability of the size of the game grid (MxN) and the number of pieces needed to win (X), it becomes computationally impossible to evaluate all possible game combinations in a finite time, especially regarding very large values of M N X. To address this challenge we have therefore introduced **iterative deepening**, an approach that calls the minimax algorithm with an increasing search depth, adapting to the available time. Choosing the best move is based on the last fully explored decision tree before time runs out.

2.2 Evaluation of the game situation

The second phase focused on **evaluating** non-final game situations. To do this we have developed an algorithm which, given the current state of the game table, returns a value by attributing weight to our sequences and to the opponent's sequences evaluated in all directions and provided that they are open on at least one side. The length of a sequence determines the weight given, with weights increasing based on length (for example, a sequence of 4 pieces is worth more than two sequences of 2 pieces).

2.3 General algorithm optimization

The third phase involved overall **optimization** of the algorithm to speed up move selection and enable greater depth of search in the decision tree in the same time frame.

Initially, we implemented the **alpha-beta pruning** technique, which allowed us to prune a large part of the decision tree, significantly reducing the computation time. Subsequently, we introduced a **transposition table** which is based on the concept of caching to avoid evaluating identical game situations multiple times. In the creation of this table a hashmap was used, which as we know exploits the concept of hashtable with linked lists which are converted into AVL trees when these lists become large. The generator of the keys necessary to access the elements of the table uses information on the arrangement of the pieces on the game grid, together with the current values of alpha and beta. In this way, each different game situation is associated with a unique key. By value, the evaluation carried out through alpha-beta pruning relating to the node in the decision tree is stored.

However, the most significant optimization was the implementation of a lightweight **preevaluation** heuristic, which further improved the efficiency of alpha-beta pruning allowing us to reach very high heights with iterative deepening. This pre-evaluation was applied in two moments:

1. Firstly, an ordering of the first n calls to alpha-beta was inserted (with n referring to the number of available columns in which the next pawn can be inserted), which uses the previous iterative deepening step as a pre-evaluation. To do this, the value of each column is saved at each iteration within a priority queue, in order to be able to exploit it in the next step, when the height of the decision tree has been increased.

2. Secondly we inserted a pre-evaluation within the alpha-beta pruning itself, evaluating, before making the recursive calls, the situation around the cell where you want to place the next checker. This heuristic considers the favorable sequences that are expanding and the enemy ones that are blocking, providing weights based on their length.

3 Approximate evaluation of the computational cost

To obtain a rough estimate of the computational cost of our algorithm, we will carefully examine each of the steps involved in its operation.

Let's start by analyzing the cost of the **iterative deepening** method, which allows us to explore the decision tree progressively, increasing the height at each iteration. The computational cost of this method is mainly influenced by the last complete execution of the iterations that occur within it, in which a maximum height is reached. This figure, as previously mentioned, is extremely difficult to estimate as it depends on two key variables: the maximum time during which a choice can be made and the specific performance of the computer on which the algorithm is run.

Let's now analyze the cost of **evaluating** a specific game situation in order to attribute a value to it. This heuristic checks each cell of the game column in order to check how many and which sequences of pieces are present in it, therefore this evaluation assumes a cost of $O(MN)$. This must then be multiplied by each node of the tree at which the evaluation is carried out, which without particular optimizations we know corresponds to the number of leaves of the decision tree, i.e. $O(N^{MN})$. We therefore found that the cost of evaluating the game tree is equivalent to $O(MNN^{MN})$.

Let us now analyze the cost of implementing the various **optimizations** used, and then only at the end evaluate the advantages that these have brought and consider the sum of these in the total computational cost.

As regards the use of the **transposition table** this was achieved efficiently, using a hash table to carry out search and insertion operations, which have a cost $O(1)$ in the average case. We also know that each of the insert and remove operations in the hash table also features the generation of a key given a certain game situation. The method that carries out this operation, i.e. generateKey, takes into consideration all the cells of the game table in order to build a unique key, therefore the cost of using this method, which dominates each insertion or removal in the hash table, is (O(MN). Since each node of the tree is inserted into or extracted from the transposition table, the total cost of using this data structure is $O(MNN^{MN})$, agreeing with the cost of evaluating the tree.

Finally, we evaluate **pre-evaluation**, which has been integrated to order the node evaluation in the best possible way, in order to improve the effectiveness of alpha-beta pruning. The cost of using this optimization includes the cost of priority queuing and dequeuing every node in the tree except leaves, which are evaluated directly and do not need preevaluation. We therefore know that these operations have a cost of $O(NlogN)$, since the maximum number of elements that are inserted into the priority queue corresponds to the number of columns on which the next checker can be inserted. For each insertion in the queue there is also a call to the preEvaluate function, which carries out an analysis of the sequences adjacent to the cell in which you want to insert the next checker, therefore the cost of this method is $O(X)$, where Christmas We therefore obtain that the computational cost of using preevaluation is $O(NlogN\sqrt{M^2+N^2}N^{MN-1})$ $= O(logN\sqrt{M^2+N^2}N^{MN})$

In the **worst case**, taking into account all the evaluations we have carried out, the overall computational cost of our algorithm is estimated at $O(MNN^{MN} + MNN^{MN} + logN\sqrt{M^2+N^2}N^{MN}) = O((MN + logN\sqrt{M^2+N^2})N^{MN})$, i.e. the sum of evaluation costs with minimax, transposition table and pre evaluation.

In the **optimal case**, however, the implemented optimizations lead to a significant improvement, allowing the evaluation of a large part of the decision tree to be skipped. As regards the improvement obtained from the implementation of the transposition table, the latter is very difficult to estimate, but what can instead be evaluated is the optimal improvement brought by the preevaluation. In fact, the latter, in the case in which it allows the nodes of the tree to be ordered so as to always evaluate the best subtree first, brings, as seen in the lesson, a quadratic speed up on the number of nodes of the decision tree, the which therefore becomes $O(\sqrt{M^{MN}})$. We can conclude that the overall computational cost of our algorithm in the optimal case is estimated to be $O((MN + logN\sqrt{M^2+N^2})\sqrt{N^{MN}})$.