Translated from Italian to English - www.onlinedoctranslator.com

ASD2023 Project Report

Alma Mater Studiorum - University of Bologna

Erik Dervishi
matriculation number: 0001069599
institutional email: erik.dervishi@studio.unibo.it

Gian Luca Montefiori
matriculation number: 0001070166
institutional email: gian.montefiori@studio.unibo.it

Piero Mongelli
matriculation number: 0001070630
institutional email: piero.mongelli@studio.unibo.it

AY 2022-23

Index

1. Introduction

The project is based on a generalization of the force 4/connect 4 game. The difference with the original game is that the match will take place in a game matrix M×N with M rows, N columns and X consecutive pieces (hence Connect X) to connect in order to win. The pieces can be connected to each other vertically, horizontally and diagonally. Each player's turn consists of inserting a token into a specific column, with the aim of concatenating as many as possible. The goal of the project is to create a bot capable of playing and winning ConnectX games against other bots. At each turn, therefore, he will have to choose a column from those available in which to insert a token. The only constraint that is placed regarding the turn is the decision time, because the choice must be made within a given TIMEOUT, beyond which the turn will be considered invalid.
To always make the best choice and win as many games as possible, it is necessary to predict your opponent's moves and act accordingly.

2 Implementation choices

2.1 SelectColumn()

The goal of selectColumn() is to choose a column among those available in which to make a move.
To choose the best move, a search strategy called Iterative Deepening is used, based on a minimax algorithm with alpha/beta pruning.

2.1.1 Iterative Deepening

The basic idea is to make the most of the time available for the shift, in order to weigh the choice so that it is the best possible.
To apply this idea the choice fell on iterative deepening. This technique performs alpha/beta pruning repeatedly, but increasing the depth with each iteration until the number of free cells at that moment is reached (i.e. the whole game is predicted) or until a win is achieved. Each time the best move up to that moment is saved, and when time is running out, the one found last is returned.

The only disadvantage of this strategy is the fact that at each iteration you must also recheck nodes already seen in the past.

2.1.2 Alpha/Beta Pruning

To evaluate and predict moves that could happen in the game we use the Alpha/Beta Pruning algorithm (highly optimized variant of Minimax). The possible moves in a turn can be seen as nodes in a tree, where the depth of the tree indicates the turn number.
With the use of a minimax the amount of computation would be too expensive, because it would have to explore too many nodes to be able to return a reliable evaluation within the time limit.
For this reason, the Alpha-Beta pruning algorithm is used, which significantly reduces the number of nodes to visit, this is possible thanks to a pruning mechanism, which ignores sub-trees that would not influence our evaluation of the optimal move.

The visit of the possible moves starts from the central column, thanks to the middlesort function, in such a way as to first see the moves that are probabilistically the best; this choice is to have a wider pruning of the nodes and speed up the finding of the best position.
The evaluation of the current move is influenced by its depth, the more the number of depth increases the lower its evaluation is, so as to assign greater importance to victories obtained with a smaller depth or to delay defeats as much as possible.

2.1.3 MiddleSort()

This function reorders an array of integers L (array indicating the different free columns where the checkers can be added) based on a logic that positions the elements of the center at the beginning, followed by the elements that progressively move away from the center.

2.2 Heuristics

The heuristic used is based on the concept of tokens. A token is a chain of tokens that could lead to a victory. For example, in connect 4, two vertical pieces in a row are one token. However, if the opponent were to place a piece on top of it, interrupting the sequence, that chain of tokens would cease to have importance because vertically it could no longer lead to a victory. The number of tokens is saved in two arrays, myToken and oppToken. The arrays have length X-1 because single tokens do not count, but only tokens starting from length 2.

The tokens of N aligned tokens are placed at index N-2 of the array corresponding to the player (mytoken=us, opptoken=enemy). The arrays are reset and filled for each move.
With each move the entire game board is studied and the tokens updated.
The game board is explored thanks to 3 methods:
verticalCount(), which counts all vertical tokens; horizontalCount(), which counts all horizontals; diagonalCount(), which counts diagonals and antidiagonals.

The value of the respective game maps are evaluated with the following criterion:

```
publicInteger heuristic() {
            calculate_tokens();

            Integer heuristic = 0;
            for(inti = 0; i < K-1; i ++) {
                  heuristic += myTokens[i]*myTokens[i]*myTokens[i] -
                         oppTokens[i]*oppTokens[i]*oppTokens[i];
                         myTokens[i]=0;
                  oppTokens[i]=0;
            }
            returnheuristic;
}
```

Below is the AnitaMaxMin code, to count the number of vertical tokens (similar executions for the other 3 directions).

```
private voidvertical_count(Cell) {
      intcounter;
      CellState expected;
      for(intj = 0; j < N; j++) {
            if(cell[M-1][j] == FREE)//free column, makes no sense
                   check it
                  continue;
            intcell2controll = 0; for(inti = 0; i <
            M; i++) {
                  if(cell[i][j] != FREE) {
                        cell2controll = i;
                        break;
                  }
            }
            expected = cell[cell2controll][j]; counter
            = 0;
            while(cell2controll <= M-1 && expected ==
                   cell[cell2controll][j] && counter < K) { counter+
                   +;
                  cell2controll++;
            }
            if(counter > 1){
               if(expected == our){
                     myTokens[counter-2] ++;
               }
               else{
                     oppTokens[counter-2] ++;
               }
            }
      }
}
```

3 Computational costs

3.1 Cost of heuristics

The value of the heuristic is calculated by the Heuristic function, which in turn calls the various count methods:

- **Count Vertical**:**OR**(**MN**)
- **Count Horizontal**:

$$OR((M-X)(N-X)X) = OR((MN-MX-NX+K2)X) = OR(MNX-MX2-NX2+X3) = \mathbf{OR(MNX)}$$

- **Count Diagonal**:

$$OR(M(N-)X) = OR((MN - MX)X) = OR(MNX - MX2) = \mathbf{OR(MNX)}$$

To calculate the final evaluation, a loop is performed to iterate through the arrays, its cost is**O(X)**.
So the cost evaluation of heuristics is**O(MNX)**

3.2 Alphabeta

As seen in class, the computational cost of alphabeta and minimax in the worst case does not improve since no pruning occurs. The number of nodes depends on the number of possible moves in a turn raised to the number of turns. The number of moves depends directly on the number of columns (N), while the number of turns is the depth of the tree (p), so the computational cost is**O(No$_p$)**. Furthermore, for every call the**middlesort**which has a cost directly proportional to**O(N)**, therefore the total cost of**alphabeta**will be**O(No$_p$)**.
Instead, if the nodes were to be evaluated, its cost would be**O(MN$_p$X)**

3.3 SelectColumn

In the selctcolumn code there is a while that iterates for the number of maximum depths it can explore in 10 seconds. Inside the loop there is also the call to alphabeta, so the total cost is**O(MN$_p$XP)**

3.4 Summary table

Functions	**Costs**
Count vertical	**O(MN)**
Count horizontal	**O(MNX)**
Count diagonal	**O(MNX)**
Heuristic	**O(MNX)**
Middlesort	**O(N)**
Alphabet	$O(MN_pX)$
Selectcolumn	$O(MN_pXP)$

4 Idea for future implementation

4.1 Saving moves/game maps

To further improve the maximum explorable depth and the speed in calculating the optimal move, a hash table should be added, which allows you to save the current map and subsequently also its value.
To do this, one of the best known methods is that of the Zobrist map, a technique used in the programming of board game algorithms, such as chess.

The Zobrist hash function involves the use of a hash table and pre-computed random numbers, known as Zobrist keys. Whenever the game state changes, an XOR operation is performed between the current value of the hash table and the Zobrist key associated with the state change.
This technique is used to generate a unique hash value for each game setup, allowing for quick verification of previous positions when searching for a game algorithm.
The main goal is to improve performance in locating previously explored locations when searching for matches. To better understand how it works we recommend the followingvideo.

5 Sitography

The idea of implementing heuristics is inspired by the followingpdf.
Idea of iterative deepening:link.