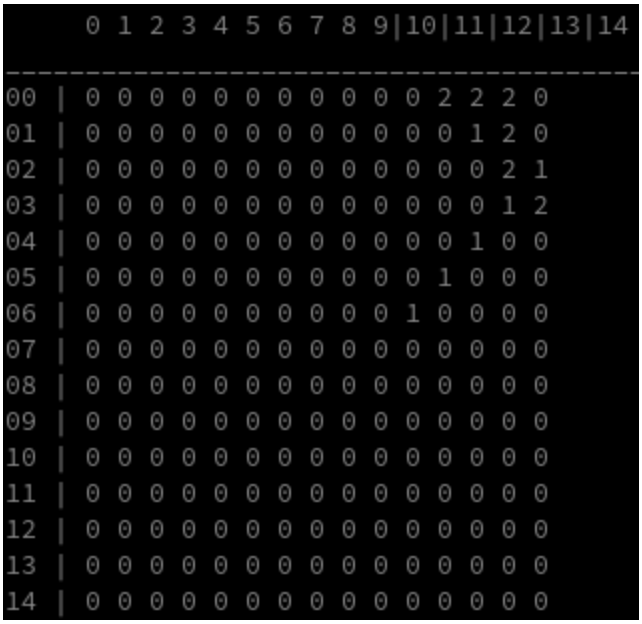
Patryk Stroński 242564

Lab 3 Artificial Intelligence

Gomoku game – duel between player and computer



Introduction

The gomoku is a simple 0-sum game between two players. The game is taking place on a 15x15 board. I utilized Rust programming language to implement minimax algorithm with alpha-beta pruning. The interface is very simple – it is a command-line one. The 1's denote player's moves whereas 2's – computer's. The user inputs the numbers like this:
4,5
Where 4 denotes position on Y axis, whereas 5 – on X axis (it's reversed comparing to cartesian coordinates).
After first of the rivals (computer or player) scores 5 points in a row, the game is finalized and result, who won is displayed.

Decision tree implementation

The decision tree is implemented using simple Structure TreeSegment – which takes the *coordinates*, *gain*, *leaves* and *minimize_leaves* fields. *Coordinates* takes x and y coordinates of the proposed field for which we calculate moving. *Gain* is the amount of points that the computer will benefit from the move. It's important to note that *gain* can be negative and is negative in case of the opposer. This is because, we assume that each good player's move is weakening the computer. This means, good move is a negative score for the player – from the logical point of view.

```rust
pub struct TreeSegment {
    pub coordinates: [usize; 2],
    pub gain: i8,
    pub leaves: Vec<TreeSegment>,
    pub minimize_leaves: bool,
}
```

The *leaves* field is responsible for holding all deeper branches of the tree. To one node there can be multiple TreeSegments attached as player might respond differently to the move.
Minimize_leaves is a property responsible for defining whether the nodes are to be minimized or maximized. In fact this means whether the leaves will contain info about computer possible moves or of the opponent. The best move of the opponent is the one that scores the least points (this is because points scored by the opponent are non-positive). This move is picked or the first one that scores the biggest value – less or equal -5.
Computer picks the move that scores the biggest value. This is chosen based on the minimax algorithm. The nodes are recursively evaluated and best ones are taken into account.

Alpha – beta pruning

This is a simple concept saving much of the performance by not evaluating all nodes. Alpha is used by the computer player, whereas beta is for the predictions of the human opponent. The pruning makes it possible to roll out all the nodes that doesn't need to be evaluated. Alpha is the best gain computer can guarantee to itself, whereas beta is the worst gain that the user can achieve. We decide to stop computation when:
1. For computer `beta <= alpha`
2. For opponent `beta >= alpha`

Outcomes

After implementing alpha-beta pruning into the algorithm it has become rapidly faster than without it. Without alpha-beta pruning having the tree depth equal 4 was the maximal, that made it possible to play the game. The computation time of bigger depths appeared to be extremaly long. The sample times for several games [ms]; depth 4:

17595, moves: 5, avg: 3519
4430, moves: 4, avg: 1107
36065, moves: 7 avg: 5152

This approximates to: 3259 ms

For the same depth but with alpha-beta pruning:
0, moves: 4, avg: 0
47, moves: 7, avg: 7
17, moves: 4 avg: 4
The difference is huge here. The approximate thinking time is equal to 4ms!

The time is even good after the depth is enlarged to 10:
49, moves: 10 avg: 5 – this time computer won
68, moves: 7 avg: 10
5 moves: 6 avg: 1
The good time is still maintained. Approximately, it is 5ms. This is just a bit more than the above and still far less than without alpha-beta pruning.

Concolusion

The above example shows clearly that alpha-beta pruning is extremally useful in game playing algorithms. Thanks to it, the computation time can be enhanced and the checking depth can also be enlarged. Moreover, after enlarging the depth, computer becomes better player what can be seen in the moves: computer makes more moves which means I also had to make more moves and the game had more turns. This implies the computer was a bit harder to defeat.
The depth therefore here can define the difficulty level. The less deeper the computer is checking the moves, the easier it is to beat it.