**CS 640 Programming Assignment 3 Report**

**AI Game ——** **4×4×4 Tic-Tac-Toe**

November 23, 2019

**Teamwork**

|  |  |  |  |
| --- | --- | --- | --- |
| **Team members** | **Kerberos** | **BU ID** | **Works** |
| Ziqi Tan | ziqi1756 | U 88387934 | Design data structure and algorithm.  Coding. |
| Kaijia You | caydenyo | U 44518396 | Coding and software testing.  Write documentations. |
| Tian Ding |  |  | Coding.  Adjusting parameters. |

The rest of this report is organized as follows. First, we review the assignment requirements. Second, we provide insight into the 4×4×4 Tic-Tac-Toe Game mostly based on [1]. Third, we go through the skeleton code and discuss our methodology. Finally, we discuss our test result. Our strategy can easily defeat the algorithm with simple defend and attack strategy and perfectly defeat random algorithm.

**Assignment Requirements**

In this assignment, we are required to implement an AI 4x4x4 cubic tic-tac-toe game by using minimax and alpha-beta pruning method which drive our AI make decisions as beneficial as possible. We should try our best to modify our algorithm and beat AI implemented from other teams for extra credits.

**Insight into** **4×4×4 Tic-Tac-Toe Game**

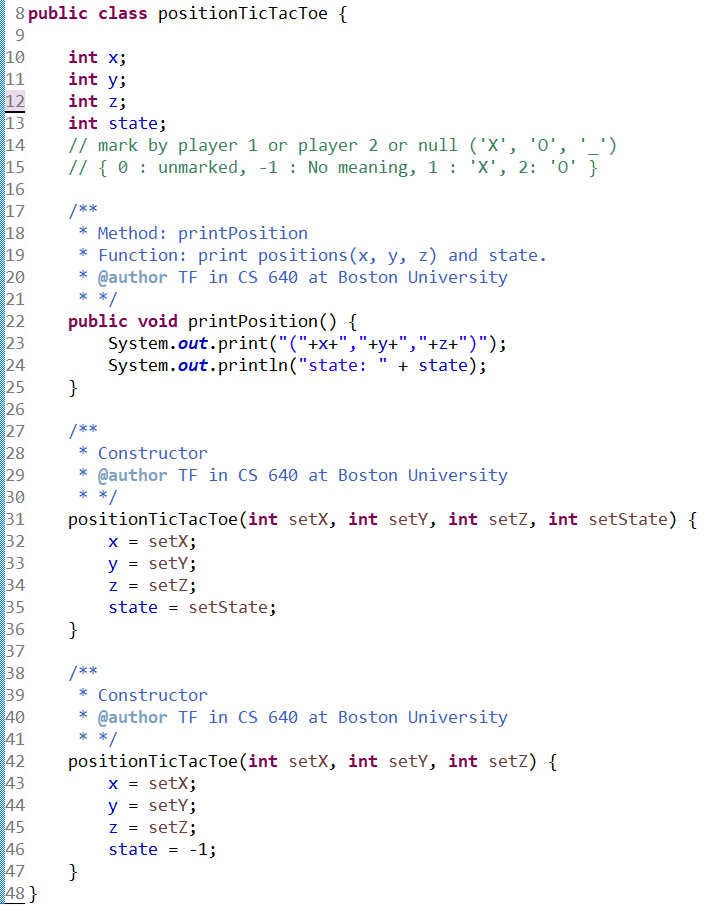
According to [1], the first player in 4×4×4 Tic-Tac-Toe Game can always force a win. Additionally, no draw exists.

**Skeleton Code Quick Review**

**runTicTacToe.java** serves as the game engine. The **run()** method is a critical part. It should be recognized that the decision algorithm (myAIAlgorithm(board, player)) will be called many times in the game.



**Data structure**



**Methodology**

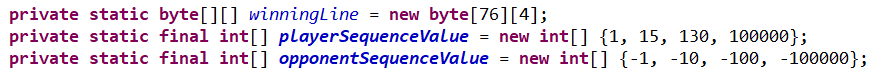
1. **Evaluation Function**

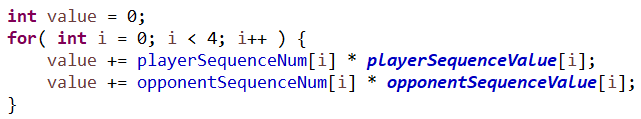
In every board configuration, the player should evaluate his/her current situation, which is a crucial part of a heuristic process for an Artificial Intelligence. We adapt the following strategy [2].

The evaluator has a list of how many 1-in-a-rows, 2-in-a-rows, 3-in-a-rows and 4-in-a-rows each player has. Where the evaluators differ is in what they do with this information, as described below.



It assigns a positive value to every n-in-a-row the player has, and a negative value to every n-in-a-row the opponent has. An n-in-a-row is worth about an order of magnitude more than an (n-1)-in-a-row.





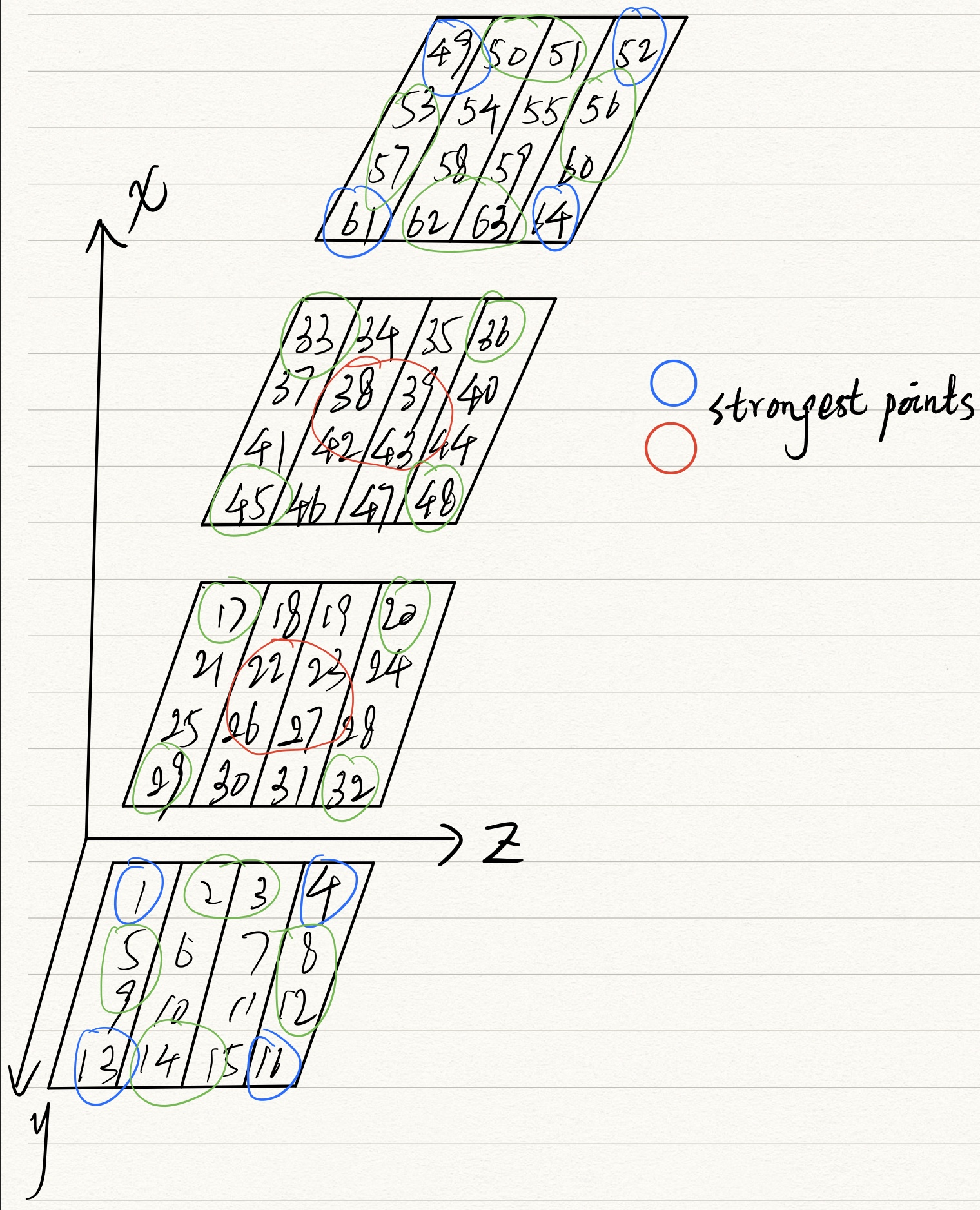
1. **Change Data Structure**

****

Without deep copy of a list of objects, this board data structure takes much less time and memory.

In the skeleton code, position in the cube is represented by an object that contains x, y, z and status which occupies a lot of memory to store when the scale of the cube becomes larger. What’s more, the for-loop in the makeMove method is terribly time cost and not necessarily. We have already known the exact node that we want to mark after doing minimax and alpha-beta pruning, however, the board is store in a List that means we must iterate the items in the List to get the position we want, which will cost O(n) in the worst case. The efficiency will be obviously better if the board is store in an array, we can mark the position simply by the subscript of the array and it only takes O(1). For this reason, we came up with the version 2 of this design.

1. **Occupy the strongest nodes as fast as possible**

****

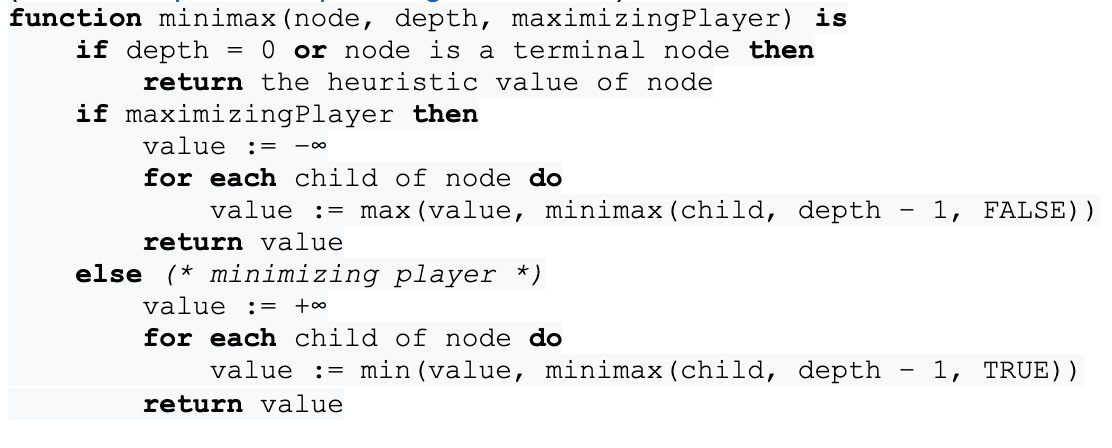
The strongest point is the point that cross by most of the winning lines. The strongest points shown in the picture has 8 wining lines respectively. It is easy to understand that the 8 nodes in the center of the cube give players more winning strategies. Each player will not win in the beginning of the game, so it is important for players to occupy the strongest nodes as fast as possible when the game begins.

1. **Winning move and force move**

As the game begins, the board will be different after players finish their turns. Player will check if 3 nodes of his/her own are filled in the same line, if yes, then player can take a winning move immediately. Otherwise player should check if 3 nodes of opponent are filled in the same line, is yes, then player has no choice but have to stop opponent from winning which is called force move.

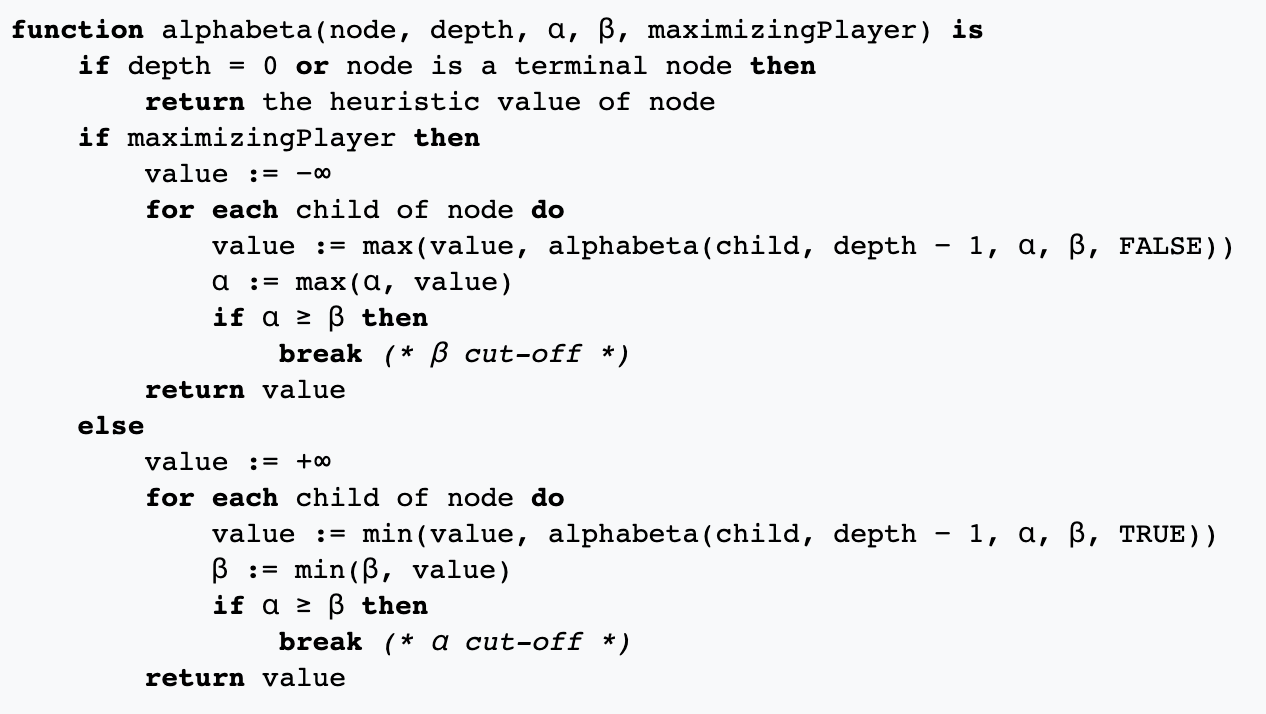
1. **minimax**

If the current chess board is not fit in the situation above then AI should make decisions by minimax and alpha-beta pruning method to maximize the its profit and minimize the opponent’s profit at the same time.



**Pseudocode of minimax**

1. **Alpha-beta pruning**



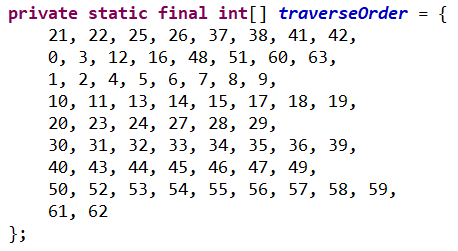
**Pseudocode of alpha-beta pruning**

1. **Progressive Deepening**

Analyze game situation to depth = 1, depth = 2, depth = 3, … until time is up.

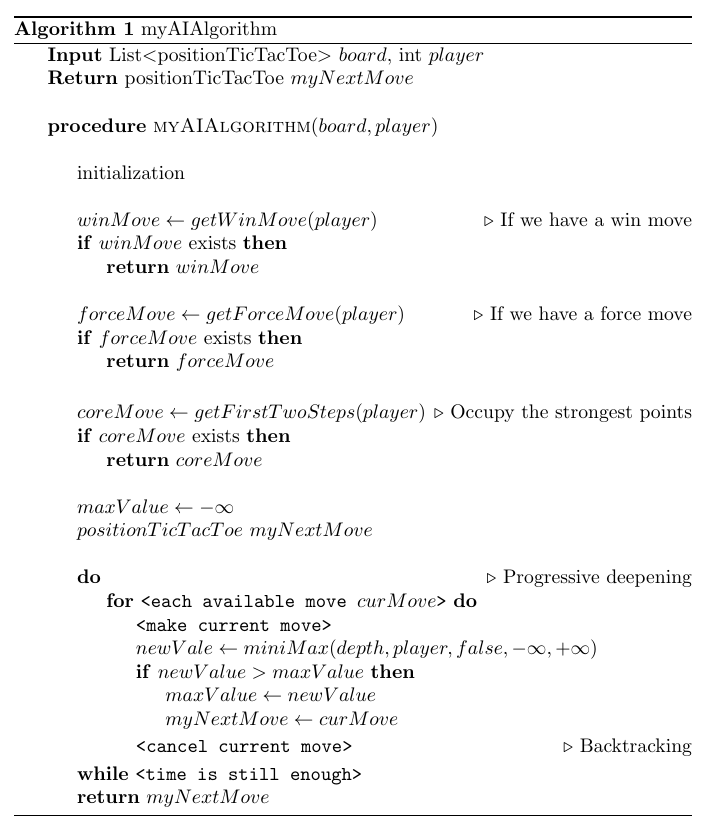
1. **Heuristic Pruning**

Define the traverse order. We traverse the strongest points first, which are obviously promising move.

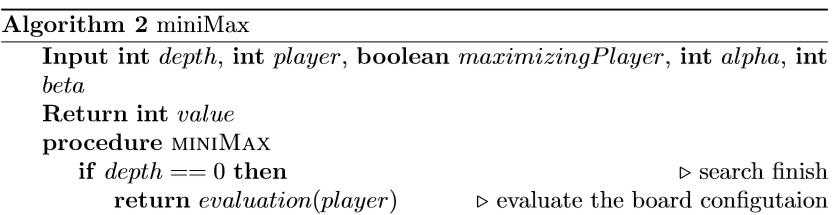


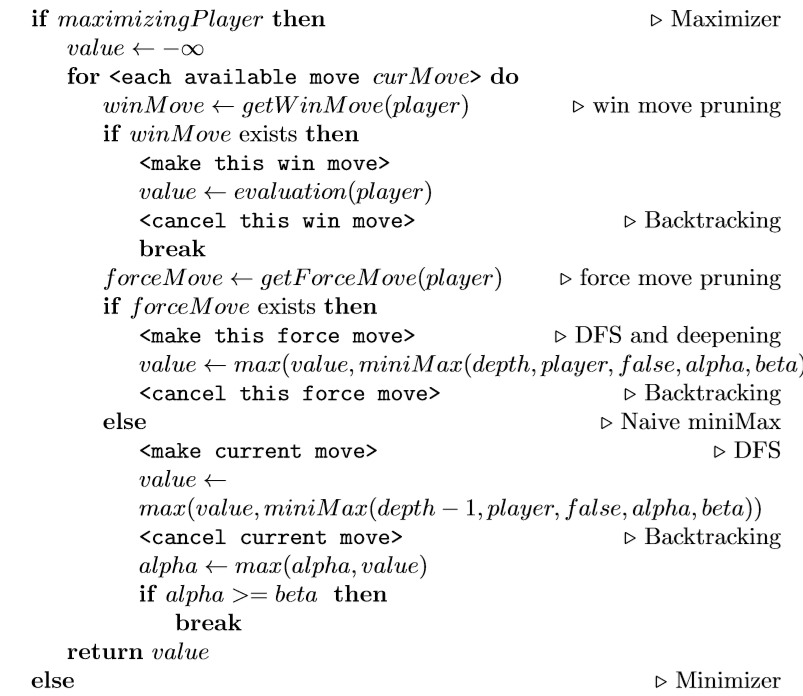
**Algorithm Design**

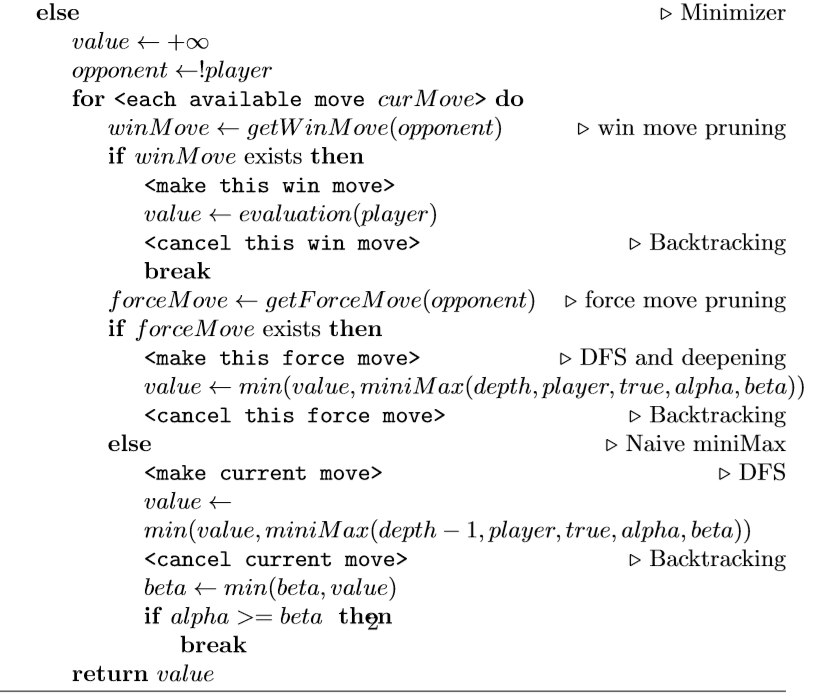
1. **Pseudocode of myAIAlgorithm**



1. **Pseudocode of miniMax**



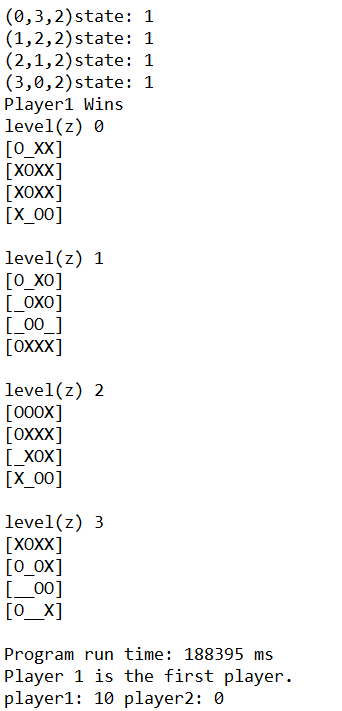
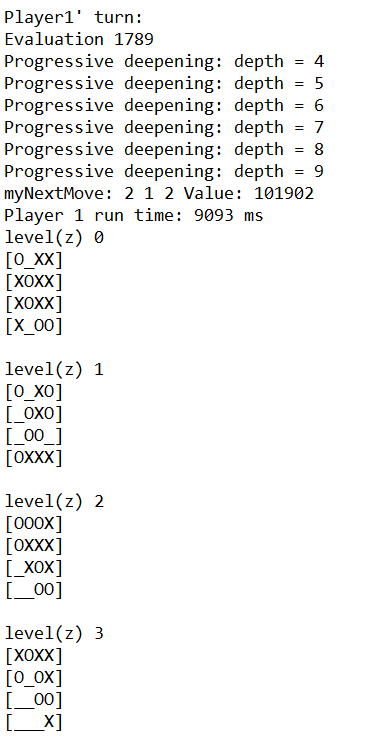




**Results**

1. **Battle between final AI version and original AI version.**

Result: final version always wins.

****

Where player 1 is the final version of AI and player 2 is our original version.

No mater who is the first player, final version always wins.

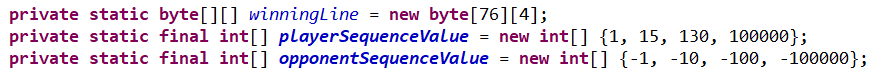
1. **Battle between final version as a first player and final version as a second player.**



Result: the second player performs better than the first player.

The reason may be that

* the second player has much more time to do progressive deepening as the number of available moves gets less and less.
* Besides, the heuristic value is not always correct.



* Additionally, in the first move of each player, they randomly choose one strongest point in the center of the cubic, which means the first step is extremely important for each player in this game.

**References:**

[1] Patashnik, Oren. "**Qubic: 4 × 4 × 4 Tic-Tac-Toe**." Mathematics Magazine 53, no. 4 (1980): 202-16. doi:10.2307/2689613.

[2] **3D Tic Tac Toe Algorithms** - Rochester CS

<https://www.cs.rochester.edu/u/brown/242/assts/studprojs/ttt10.pdf>