**CS 640 Programming Assignment 3 Report**

**AI Game ——** **4×4×4 Tic-Tac-Toe**

November 20, 2019

**Teamwork**

|  |  |  |  |
| --- | --- | --- | --- |
| **Team members** | **BU Kerberos** | **BU ID** | **Works** |
| Ziqi Tan | ziqi1756 | U 88387934 | Design data structure and algorithm.  Coding. |
| Kaijia You |  |  | 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 deeper 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. Besides, two versions have been developed but the final version is just slightly better than the other one. 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

**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

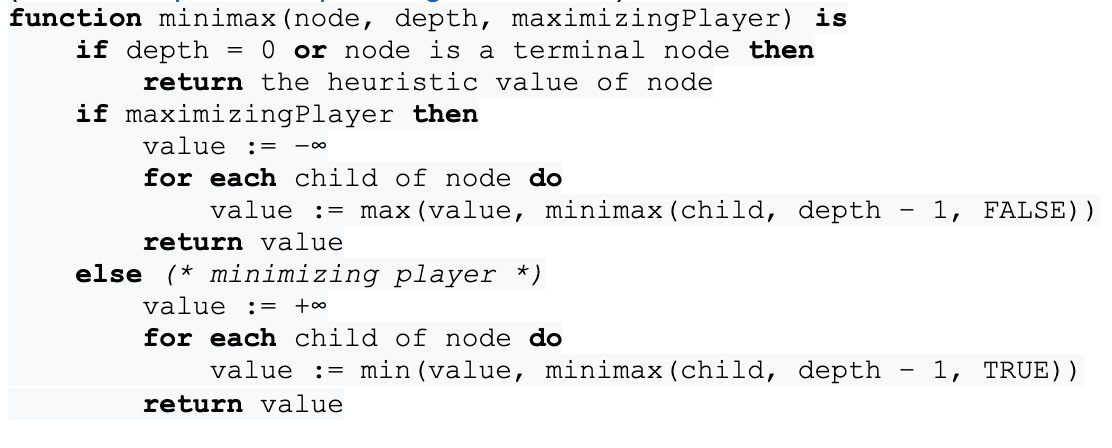
**Methodology**

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.

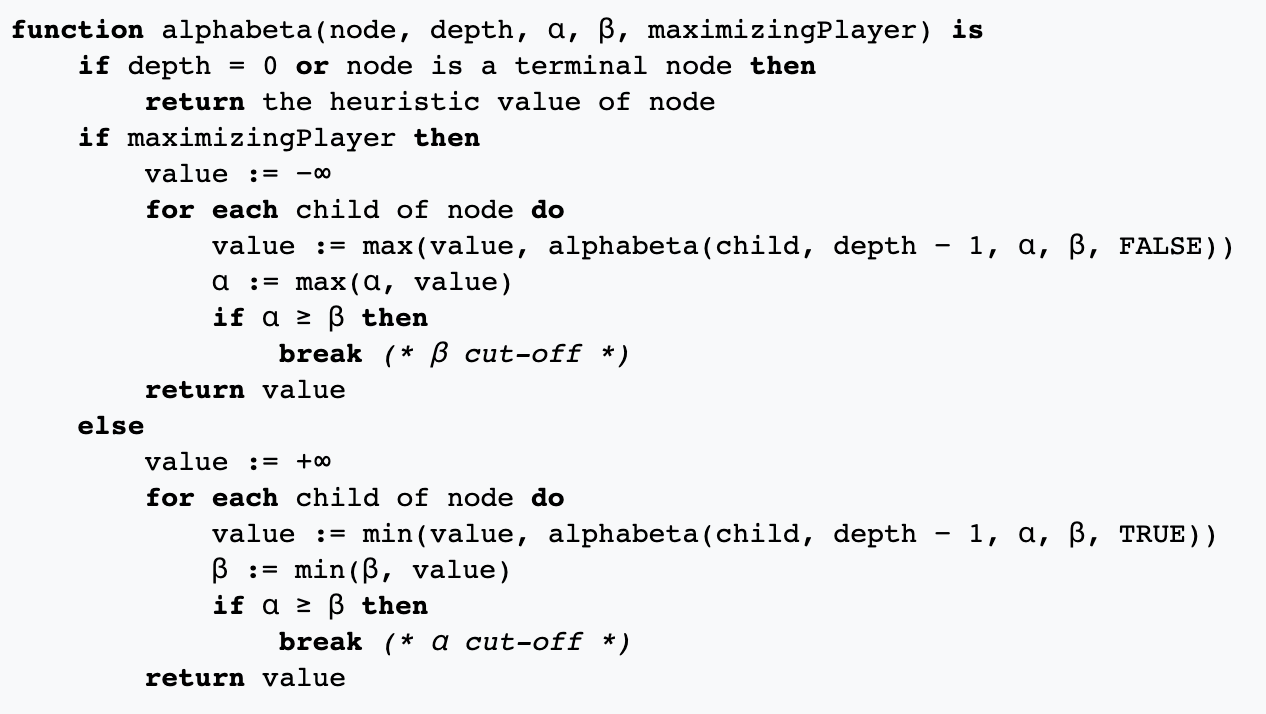
1. **Change Data Structure**
2. **Occupy the strongest nodes as fast as possible**
3. **Winning move and force move**
4. **minimax**

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**
2. **Heuristic Pruning**

Algorithm Design

**Implementation**

Version 1

Adopt the original data structure and naïve minimax with alpha-beta pruning.

Final version

**Method and Implementation**

**Original version (Using postionTicTacToe data structure):**

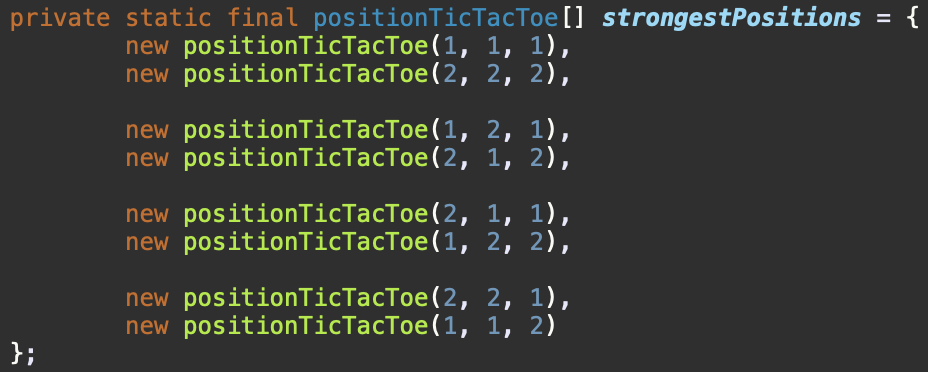
1. **Winning move and force move**

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.

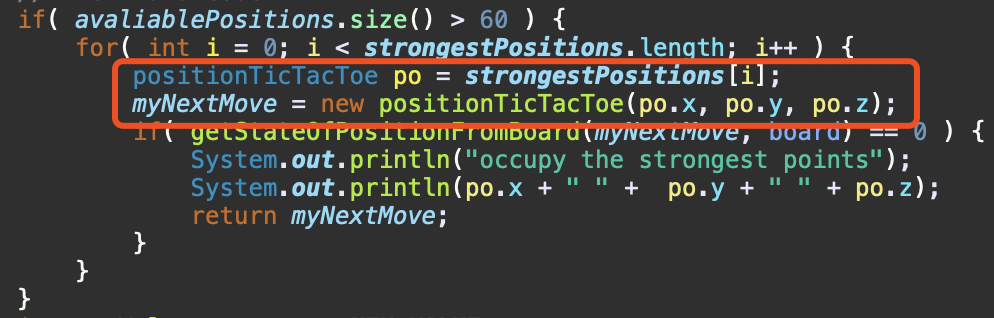


As for winning move, there are 76 winning lines in total. What AI should do is to iterate all of these lines and see if any 3 nodes of its own fit in the situation and then make a winning move. Force move is similar to the winning move.

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

****

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.



**Reflection**

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.

**Better version (Use byte array data structure):**

**Results**

**Battle between version 1 and final version**

**Battle**

**Discussion:**