Summation
May 2010, pp. 5-8
http://ripon.edu/macs/summation

Artificial Intelligence: Implementing 3D Tic-Tac-Toe in C++

Bradley D. Bogenschutz
Department of Mathematics and Computer Science – Ripon College

***Abstract*—This paper will discuss how we implementated an intelligent game of 3D tic-tac-toe in C++ using the minimax algorithm and alpha-beta prunting.**

***Key Words*— alpha-beta pruning, artificial intelligence, heuristic, minimax, ply.**

I. Introduction

IMPLEMENTING the game of 3D tic-tac-toe in C++ brought upon challenges. In this paper, our group's implementations decisions for a 4x4x4 version of tic-tac-toe will be discussed along with support of why we chose them. The techniques used to generate an intelligent version of this game have been described in the first paper of this series. These techniques include minimax alorthim, alpha-beta pruning, and heuristics. However, the focus will change from what these techniques are in a general sense to how they specifically affect implementation decisions for our program.

The implementation of our program can be broken into two main parts: creating an intelligent game and C++ coding. The implementations of creating an intelligent game of 3D tic-tac-toe are divided into sections. These sections that will be discussed are the future of the board, the heuristic, the minimax algorithm, and alpha-beta pruning. On the other hand, the implementation of the C++ code is less complex then making the program intelligent, but without it there would be no program. So, that section will discuss ways we utilized C++ code to display the board and represent the board. These two main parts talk about the implementation of our specific program, but an understanding of the game of 3D tic-tac-toe is needed. So, the paper will begin with information about what 3D tic-tac-toe is in order to acquaint the reader with our program.

II. 3D Tic-Tac-Toe

The 4x4x4 tic-tac-toe game is very similar to traditional tic-tac-toe but there are more possible places to make a move because of the extra dimension. The goal in playing a 4x4x4 game of tic-tac-toe is to be the first player to get four pieces in a row. The first to get four pieces in a row is the winner.

A. 4x4x4 Tic-Tac-Toe

The game of 4x4x4 tic-tac-toe can be visualized as four 4x4 grids stacked vertically on top of one another. The four grids allow for more possible wins than the traditional game of 3x3 tic-tac-toe. The number of possible wins is greater because instead of being able to win by filling all possible moves in a vertical, horizontal, or diagonal line on a grid, users are able to make those moves plus the similar lines but this time going through each grid. Examples of every possible move can be seen in figure 1 if you rotate each board and shift the line across the board. Not every possible move can be shown in a simple image because there are 76 possible wins. With the 76 possible wins there are 64 possible places to make a move in order to generate a win. The number of possible moves and wins will be reduced every time someone makes a move. The reduction of possible wins every time someone makes a move makes this an interesting game to program intelligently.

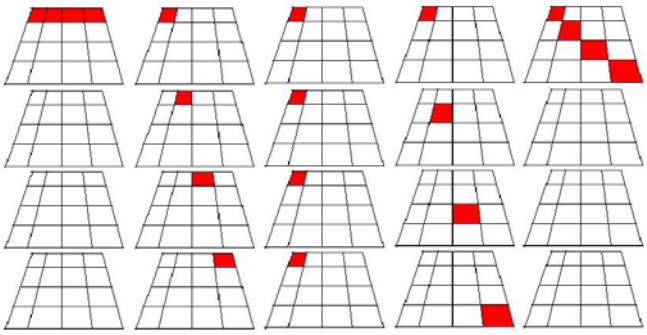

Figure 1. Illustration of possible wins in 4x4x4 tic-tac-toe.

B. Why a 4x4x4 Board Size

Why would we choose to make a computer game of 4x4x4 game of tic-tac-toe when the traditional tic-tac-toe board is a 3x3 grid and not a 4x4 grid? The reason we chose to not go with a 3x3x3 board is because an advanced player will have won the game after both players made a combined total of seven moves. This means if we wanted to create a program to play a 3x3x3 version of tic-tac-toe we would have it go through an algorithm that looked at the user move and made its move without using any artificial intelligence. This advantage to an advanced player is greatly dependent on the

Work presented to the college March 24, 2010 as the second paper in the *3D Tic-Tac-Toe* series.

B. D. Bogenschutz is pursing his baccalaureate degree at Ripon College, Ripon, WI 54971 USA (e-mail: bogenschutzb@ripon.edu).

fact that a placement of a piece on the center most spot of a 3x3x3 board would eliminate approximately half the possible wins for the other player. For this reason, we chose to consider a 4x4x4 size board for our program. The 4x4x4 tic-tac-toe was proven that an expert player can force a win. However, it is still challenging to an average player.

We did not rule out a 4x4x4 tic-tac-toe board size for our program but we did look into creating a game using a 5x5x5 tic-tac-toe board size. The game of 5x5x5 increased the possible moves from 64 in a 4x4x4 board to 125 possible moves. Because of the increase in possible moves, the game was no longer to simple for an advanced player. However, the problem is now trying to program an intelligent game for this board size. When we program an intelligent game of 3D tic-tac-toe the computer looks at the future of the board by generating all possible future moves up to a certain number of moves ahead of what the board is currently. The searching and generation of the future moves on a given board grows exponentially when the player looks more moves ahead. However, looking more moves ahead will generate a better move because the player can predict the future better. By taking a look at 4x4x4 board verses a 5x5x5 board you will notice a large difference in the number of possible states of the board the program will generate. When only looking three moves ahead, a 4x4x4 board will generate roughly 262,000 future board states compared to roughly 1,953,000 future board states for a 5x5x5. This number is greatly larger and affects the runtime and memory consumption needed to run the program. With this being said, we would run into problems generating an intelligent computer move. Thus, we decided to create the computer game of 3D tic-tac-toe to be of a 4x4x4 board size.

III. Implementation Decisions to Achieve Our Goals

To reiterate the goals of our program, our program should look at the future of the board in order to make a more intelligent move, minimize the amount of time it takes for the computer to process a computer move, and reduce the amount of memory consumption. Our goals will be greatly affected when programming the future of the board, implementing the heuristic, implementing the minimax algorithm, and by alpha-beta pruning.

A. Future of the Board

Looking farther into the future of the game is done by generating all possible moves for the next user and then the possible moves for the next user from the previous moves and so on. Each possible move can be described as the state of the board or a node in a tree. The states or nodes represent a current or possible configuration of game pieces on a board. However, the farther you look into the future, the more intelligently the move is. This is because the computer assumes that, with its heuristic, each move is the best possible move for either the computer or user at that current state. So ideally, I wanted to look as far ahead as possible to make the best move.

However, before we look into how far ahead we can look we must understand how we refer to the future of the board. When implementing the future of the board the computer will generate a tree, an example tree can be seen in figure 2. This tree will look ahead a fixed number of plies. A new ply is created every time the computer looks ahead another move. For example, if the computer is looking four moves ahead then the game is a four ply game. The game must look ahead an even ply. This is because we want the computer to calculate the best possible move when it is the computers move for the next state. Thus, our program can be of two ply, four ply, six ply and so on. As a programmer the game should be coded to look as far ahead as possible but maintain a reasonable response time for a computer move after the user makes a move.

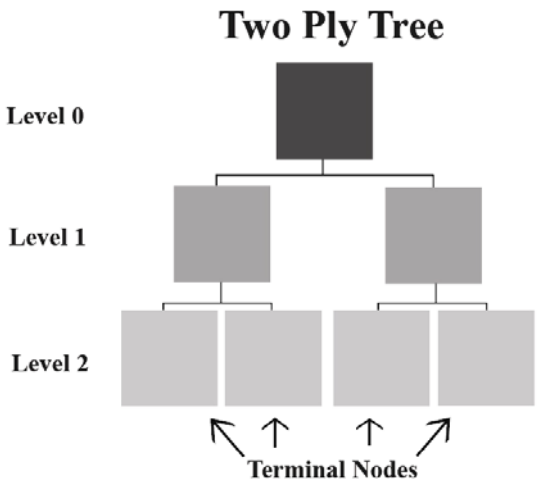

Figure 2. Example of a simple tree.

We chose to go with a four ply game. The reason we went with a four ply game instead of six ply game is because of an issue with time to generate the next computer move. When generating the next move, a tree that looks ahead four ply compared to one that looks ahead six ply will have a notice difference in response time. This is due to the fact that the number of possible states that need to be generated and taking into consideration grows exponentially when going from one ply to the next ply. A four ply game will have to generate and search through roughly sixteen million states which are a lot less then that of a six ply game which generate and search through roughly 68 billion states. Thus, our program looks ahead four ply.

There is a problem with coding a program that looks at the future of the board. The problem arrives when the board becomes almost full. Since the heuristic is calculated at a terminal node, the bottom of tree, the program can return a move of position -1. This is a problem because each position on the board is represented by a numbers between 0 and 63. If this occurs, the heuristic is calculated for the current state and the computer's next move is made based upon the calculation of the current state. This move is still intelligent because the errors will only occur when the computer cannot look 4-ply ahead and when the computer realizes the game is a draw. So,

there is no need to look far into the future of the game in order to make an intelligent move.

B. Heuristic Implementation

The heuristic is calculated to find the best possible move at the current state of the game. Our heuristic is calculated by searching through each of the 76 possible wins, then counting the number of user and computer pieces in that possible win, and finally it plugs the counts into an equation and updates the pieces affected by that possible win. The equation is calculated for each piece on the board and can be written as

Heuristic =
((10000*nolw3cp)+(100*nolw2cp)+(10*nolw1cp)+(1*nolw0cp)) –
((10000*nolw3up)+(100*nolw2up)+(10*nolw1up)+(1*nolw0up)).

The nolw#cp stands for the number of lines with # computer pieces. Simular with nowlw#up, except it is user pieces instead of computer pieces. In figure 3, I have simulated part of a game of red(opponient) and green(home) pieces and all of the blank spaces will have a heuristic calculated for. When calculating the heuristic for piece number one, you will get a heuristic of 200 by using the equation above. You can also calculate the heuristic for piece number two and number three. Number two has the heuristic of -80 and number three has the heuristic of -20. The program will then search through the entire board to find the best move. In this case with only three heuristics calculated, number one would have the best chance of winning. This is a very affective way of calculating our good heuristic. The program is very simple and involves very little runtime compared to other ways of calculating a valid heuristics.

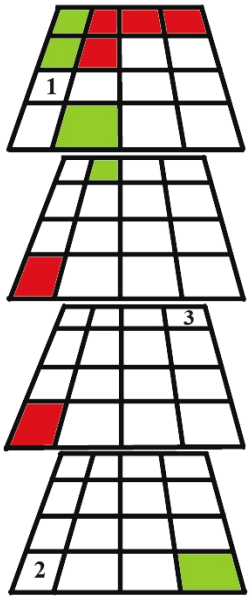

Figure 3. Illustration of a simulated game.

C. Minimax Implementation

Minimax algorithm implementation is implemented in our program by recursion. In order to implement the minimax algorithm you need generate a tree with all possible moves. The problem with generating a tree is the amount of memory needed to store the information. Since our program is a four ply game we will need to generate a tree that has roughly sixteen million nodes. Storing all sixteen million nodes would become a problem. Therefore, we chose to represent the tree by using recursion. Using recursion on a four ply tree will allow for the computer to only store at most five nodes in memory at one time. This is a lot smaller then storing about sixteen million nodes into memory at one time. The use of recursion allows the program to generate possible moves down one branch of a tree until it reaches a terminal node. When the program reaches a terminal node it calculates the heuristic and then returns the best move to the previous node. The previous node then calls the next node until a terminal node is reached. This process repeats until the computer has looked at all the tree's nodes. At that point, the computer will know the next best move for the game and will make that move.

D. Alpha-Beta Pruning Implementation

Alpha-beta pruning is a way of eliminating unnecessary nodes. The elimination of the unnecessary nodes will reduce the amount of runtime because not all the nodes will have to be looked at. In order for a programmer to implement alpha-beta pruning within the recursive part of the program, the programmer will need to keep track of the best move for the computer, alpha, and the best move for the user, beta. The program will also have set to see if it is calculating the best move on a min or max level of the minimax algorithm.

So, when implementing alpha-beta pruning, the program will first look to see if it is at a max or min level in the minimax algorithm. If it is a max level the program will compare alpha and beta. If alpha is greater that or equal to beta then the program knows the possible children in the game tree will not be better than a different node that has its heuristic already calculated. Therefore we can utilize pruning by not doing the computations for the children nodes of that branch since, we know that a better move is not possible due to the nature of how alpha-beta pruning works. Simular, a min level is handeled in the same fashion as a max level but it tests to see if beta is less than or equal to alpha. This technique is a simple test in the programming code as well as keeping track of alpha and beta. However, when certain nodes and their children are not looked at, they will not result in a better move by the computer and the number of computations is reduced. Thus decreasing the amount of runtime it takes for the computer to make a move.

IV. Implementation in C++ Coding

Although the majority of this program's complexity is in the code that makes the program artificially intelligent, the program is still being coded in C++, which required me to make other coding decisions. When coding the program of 3D tic-tac-toe we had to determine how to represent the board, return the correct value to a certain function, how to display the board to the users, and also how to test the program for correctness.

When deciding how to represent each piece on the board, we needed to decide what kind of array we were going to use in order to contain the 64 possible moves. We chose to use a one-dimensional array because of simplicity of programming. Our program required a large amount of searching to test, find, and output data. In doing these operations we used FOR loops. By choosing to represent the board as a one dimensional array the program did not require nested loops. This made it easier to program and find errors in our code.

However, when choosing how to have a user input their next move, we made our program ask the user to input the position they want in a row, column, and level format. This type of entry is easier for a user to determine where a spot is on the board.

When displaying the game we had to consider the idea of how to represent a three dimensional game onto a two dimensional surface. So, in our C++ version of the game we display the board by outputting four 4x4 grids and place them in a vertical line. This output is fairly simple to read for being displayed in two dimensions.

Another problem we had when making this program was returning the correct value of the best possible move back to the main program. Since we used recursion to simulate the tree, we needed to return the value of the best move within the recursive function. However, when the recursive function was done searching through the tree and it was time to return the best move back to main function, we did not want our program to return the best value but to return the position of the best valued move. Thus, we added extra code that tested to see if the program went through the entire tree and was at the top level and if it was we returned the position of the best move.

These problems along with other coding problems were caught by testing code. Testing code was crutial in making our program work properly. This required additional code to be added to the program. This additional code usually output the values of current variables. The values were analized and corrections to the code were made based upon our analysis. Overall, the testing code allowed us to make sure our results were correct and the proper move was chosen.

V. Conclusion

The goals of the program have been obtained by creating a successful game of 3D tic-tac-toe. Also, the game was intelligent by looking at the future of the board and used a good heuristic, it used recursion to reduce memory consumption, and it used alpha-beta pruning and was four ply to maintain efficient response time.

Appendix

See link for the code:
http://ripon.edu/ academics/macs/summation/2010/supp/3DTTTcpp.pdf

Acknowledgment

I would like to thank my fellow group members, Jay Hardacre and Danny Hanson, for their assistance in the design and analysis of this project. I would also like to thank Professor McKenzie Lamb for his guidance during this project. Finally, I would like to thank the rest of the faculty in the Ripon College Mathematics and Computer Science Department for all their help and support.

References

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2nd ed. Upper Saddle River, NJ: Pearson Education, 2003.

[2] Steven Tanimoto. *The Elements of Artificial Intelligence Using Common LISP*. New York, NY: Computer Science Press, 1990. 194-202.

Bradley D. Bogenschutz was born in Appleton, Wisconsin on February 19th, 1988. He graduated from Hortonville High School in Hortonville, Wisconsin in 2006. Brad is currently studying at Ripon College in Ripon, Wisconsin to receive a bachelor degree. His field of study is in computer science.

He is currently employeed by Ripon College and works in the information technology services department. Over the past few summers he has worked for Sudexo as a painter and for Neenah Papers as a mill hand. He has also maintained the Theta Chi-Delta Omega chapter's website while at Ripon College.

Mr. Bogenschutz is a member of the Ripon College Track & Field team and Theta Chi Fraternity. He is also involved in intermeral sports. In his free time Brad enjoys to fish, hunt, snowmobile, and photography.