**1. INTRODUCTION**

This report presents a way by which computers can be programmed to learn to play classical board games against humans. The model has been implemented and tested in the domain of tic tac toe. Suggestions on how to further improve the model is also presented.

**1.1. Problem Definition**

The Project is aimed at implementation of an intelligent version of the game tic tac toe. A 3D version with each side having four positions along any dimension is expected. The game should be include learning, and should have a heuristics-based approach. Implementation of minimax search and alpha-beta pruning contribute towards making the game more intelligent. Artificial Intelligence deals with writing computer programs that can solve problems creatively. It includes techniques that imitate the step-by-step reasoning that humans often use to solve puzzles, play board games or make logical deductions. The game, being code in Java, makes it necessary to have the Graphical User Interface or GUI to be Java Applet or Java Swing.

**1.2. Scope**

The project will cover moderate to high level of programming of techniques in Artificial Intelligence such as minimax search and alpha-beta pruning. The goal is to create an intelligent game by looking at the future of the board and use a good heuristic. Concepts like irregular threads with dynamic parallelism need to be taken care of, during the implementation. This involves high complexity which increases exponentially with stepwise increase in dimension of the board. Considering all these complexity constraints, the Java code is implementation includes creation of a 4x4x4 board. The main scope of project is to create a two-player game system & calculate the best possible move for a certain situation by learning what board states are most beneficial and hence working towards those states. Thus, avoiding using brute-force method makes the game an interesting prospect.

**2. DESCRIPTION**

Artificial intelligence is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable. The field was founded on the claim that a central property of humans, intelligence can be so precisely described that it can be simulated by a machine. Artificial intelligence has been the subject of optimism, but has also suffered setbacks and, today, has become an essential part of the technology industry, providing the heavy lifting for many of the most difficult problems in computer science. AI research is highly technical and specialized, and deeply divided into subfields that often fail to communicate with each other. AI research emphasizes on the optimization of search algorithms [1].

In this project concepts of A.I. such as mini-max search, alpha-beta pruning were implemented.

**2.1. Selection of dimensions of the board**

A 4x4x4 board is chosen rather than a 3x3x3 board because an advanced player will have won the game after both players made a combined total of seven moves. Also the 3x3x3 version can be programmed easily without using artificial intelligence. This advantage to an advanced player is greatly dependent on the 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. Hence, 4x4x4 board is better and is proved that an expert player can force a win and challenging to an advanced player.

Another possible option of 5x5x5 board size increased possible moves to 125 from 64 in a 4x4x4 board. Thus, the level becomes much challenging even for an advanced player. In 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. The 4x4x4 board has large number of differences than 5x5x5 board in the number of possible states of board the program will generate. By looking three moves ahead, a 4x4x4 board generates roughly 262,000 future board states compared to 5x5x5 board which generates 1,953,000 future board states. This number is greatly larger and affects the runtime and memory consumption needed to run the program. As a result, a 5x5x5 board would run into problems generating an intelligent computer move. Therefore, 4x4x4 was accepted as the best one.

**2.2. Theoretical layout**

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.



Figure 2.1 Winning possibilities

Examples of every possible move can be seen in figure 2.1 if one rotates each board and shifts the line across the board. There are 76 possible wins and 64 possible places to make a move in order to generate a win. The possible wins are as follows:

The leftmost board shows a win wherein a player gets four of his pieces in a straight line in one plane, along the length of the plane. There are eight possible ways per plane to enforce a win in this manner, thus making a total of thirty two ways on the entire board. The second board shows a win when a player gets four of his pieces along a diagonal running across the four planes and parallel to the sides. Four such diagonals emerge from every position along each side of the topmost plane of the board. Thus, sixteen diagonals emerge in all, making way for sixteen winning possibilities. The next board shows a win by a player by placement of four of his pieces along a vertical line passing through the same position on each and every plane. As we can clearly make out, there are sixteen such vertical lines possible on the board each of which provides a distinct way of winning the game. The fourth board position shows the most interesting way of winning which is possible only in case of a 3D tic tac toe. Since four such long diagonals exist, we can have four different ways of winning by these positions. And finally, the last board shows pieces placed along the diagonal, albeit in the same plane. Two diagonals in each plane provide a total of eight winning ways. The number of possible moves and wins will be reduced every time someone makes a move. The reduction of possible wins every time one makes a move makes this an interesting game to program intelligently.

**2.3. Future of the Board**

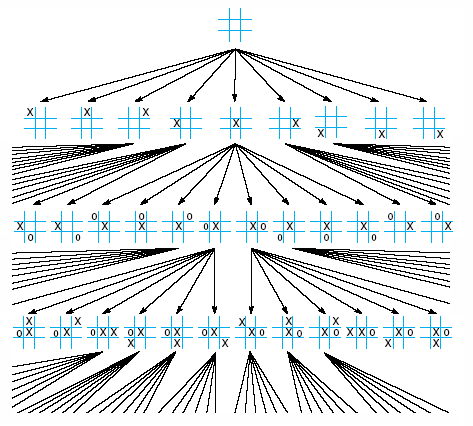
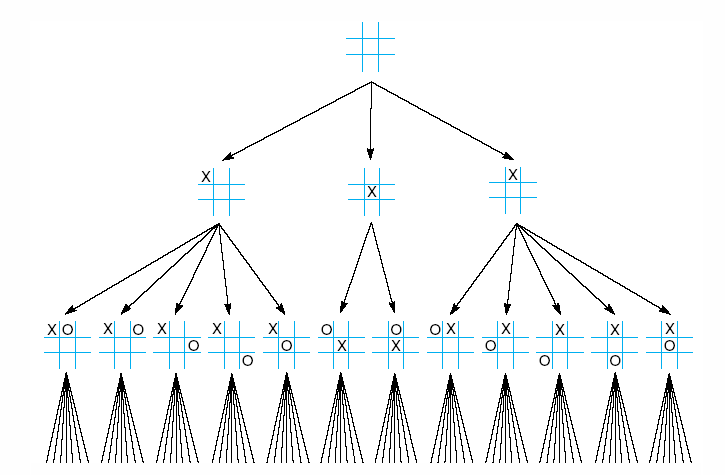
****

Figure 2.2Two-ply tree Figure 2.3Four-ply tree

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 describes as the state of the board or a node in a tree. The states or nodes each represent a current or possible configuration of game pieces on the board. However, the farther you look into the future, the more intelligent the move is. This is because the computer assumes that with its heuristics, each move is the best possible move for either the computer or user at that current state. So, the idea is to look the farthest to make the best move.

While implementing the future of the board, the computer generates a tree structure as seen in figure 2.2. 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 looks four moves ahead, then the game is a four-ply game (figure 2.3). The game must always look ahead an even ply. This is because we want computer to calculate the best possible move for its turn.

**2.4. Minimax Search**

Minimax is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss while maximizing the potential gain. Alternatively, it can also be thought of as maximizing the minimum loss. Originally formulated for two-player zero-sum game theory, covering both cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision making in the presence of uncertainty [3].

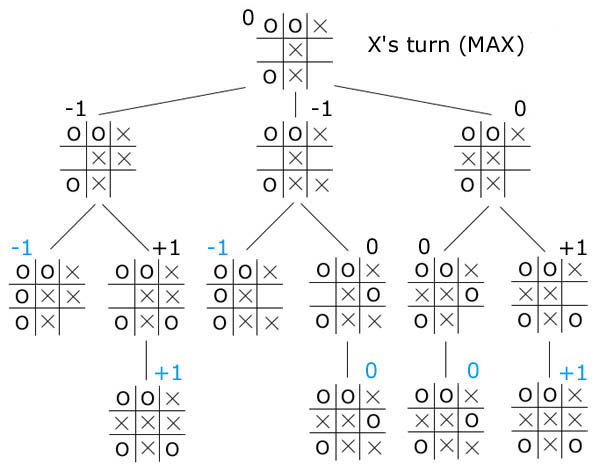


Figure 2.4 Minimax search calculations

For simplicity, consider a section of a game tree for 2D 3x3 tic tac toe. Each node represents a board position, and the children of each node are the legal moves from that position. To score each position, we will give each position which is favorable for player ‘A’ a positive number (the more positive, the more favorable). Similarly, we will give each position which is favorable for player ‘B’ a negative number (the more negative, the more favorable). In our tic tac toe example, player ‘A’ is 'X', player ‘B’ is 'O', and the only three scores we will have are +1 for a win by 'X', -1 for a win by 'O', and 0 for a draw. Note here that the blue scores are the only ones that can be computed by looking at the current position. To calculate the scores for the other positions, we must look ahead a few moves. Assuming that the opponent is rational, i.e. the opponent can compute moves just as well as we can, and the opponent will always choose the optimal move with the assumption that we, too, will play perfectly.

One algorithm for computing the best move is the minimax algorithm:

*minimax(player,board)*

*if(game over in current board position)*

*return winner*

*children = all legal moves for player from this board*

*if(max's turn)*

*return maximal score of calling minimax on all the children*

*else (min's turn)*

*return minimal score of calling minimax on all the children*

If the game is over in the given position, then there is nothing to compute; minimax will simply return the score of the board. Otherwise, minimax will go through each possible child, and (by recursively calling itself) evaluate each possible move. Then, the best possible move will be chosen, where best move leads to the most positive score for player ‘A’, and the board with the most negative score for player ‘B’.

**2.5. Alpha-beta pruning**

For a simple game like tic tac toe, it is certainly possible to search all possible positions. For a game like Chess however, the running time is prohibitively expensive. In fact, to completely search this game, we would first need to develop interstellar travel, as by the time we finish analyzing a move the sun will have gone nova and the earth will no longer exist. Therefore, all real computer games will search, not to the end of the game, but only a few moves ahead. Of course, now the program must determine whether a certain board position is 'good' or 'bad' for a certainly player. This is often done using an evaluation function. This function is the key to a strong computer game; after all, it does little good to be able to look ahead 20 moves, if, after we do, we decide that the position is good for us, when in fact, it is not [2].

However, there is an optimization we can make to the simple algorithm above that will save us a lot of searching and allows us to increase our maximal search depth. To understand the basic idea, consider the following: It is our turn to move, and we have just finished computing a move (move A) which gives us a large advantage. Now, we are attempting to analyze a second move (move B), and we discover that, in the very first response we consider for our opponent, our opponent can force a neutral position! Then there is no reason for us to continue examining move B. At this point, we know that the best we can do if we play move B is to gain a neutral position, and we already have move A which can guarantee us a better position.

To formalize this idea, we will keep track of two numbers, alpha and beta for each node we're analyzing, hence the name for this algorithm alpha-beta pruning. Alpha will be the value of the best possible move you can make, that you have computed so far. Beta will be the value of the best possible move your opponent can make, that you have computed so far. If at any time, alpha >= beta, then your opponent's best move can force a worse position than your best move so far, and so there is no need to further evaluate this move. The same pruning condition applies if you are the min player, except instead of finding the move that yields alpha, you would find the move that yields beta.

To guarantee that this algorithm returns a move, we can start alpha with -infinity (the best you can do is lose) and beta with infinity (the worst your opponent can do is to let you win), and update these values as we examine more nodes.

alpha-beta(player, board, alpha, beta)

if(game over in current board position)

return winner

children = all legal moves for player from this board

if(max's turn)

for each child

score = alpha-beta(other player, child, alpha, beta)

if score > alpha then alpha = score (we have found a better best move)

if alpha >= beta then return alpha (cut off)

return alpha (this is our best move)

else (min's turn)

for each child

score = alpha-beta(other player, child, alpha, beta)

if score < beta then beta = score (opponent has found a better worse move)

if alpha >= beta then return beta (cut off)

return beta (this is the opponent's best move)

Notice that it is to our advantage to generate the best moves first for max and the worst moves first for min; this will cause the algorithm to cut off more quickly. On average, using a good successor generator will allow alpha-beta to search to a level twice as deep as minimax in the same amount of time. Note also that alpha-beta returns the same score as minimax; it simply returns the same result in faster time.

Normally during alpha-beta, the subtrees are temporarily dominated by either a first player advantage (when many first player moves are good, and at each search depth the first move checked by the first player is adequate, but all second player responses are required to try and find a refutation), or vice versa. This advantage can switch sides many times during the search if the move ordering is incorrect, each time leading to inefficiency. As the number of positions searched decreases exponentially each move nearer the current position, it is worth spending considerable effort on sorting early moves. An improved sort at any depth will exponentially reduce the total number of positions searched, but sorting all positions at depths near the root node is relatively cheap as there are so few of them. In practice, the move ordering is often determined by the results of earlier, smaller searches, such as through iterative deepening.

The algorithm maintains two values, alpha and beta, which represents the minimum score that the maximizing player is assured of and the maximum score that the minimizing player, is assured of respectively. Initially alpha is negative infinity and beta is positive infinity. As the recursion progresses the "window" becomes smaller. When beta becomes less than alpha, it means that the current position cannot be the result of best play by both players and hence need not be explored further.

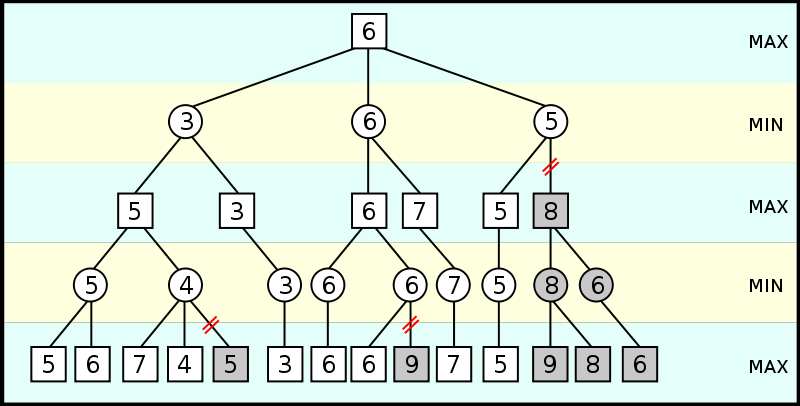


Figure 2.5 Alpha-beta pruning

An illustration of alpha-beta pruning is shown in figure 2.5. The grayed-out subtrees need not be explored (when moves are evaluated from left to right), since we know the group of subtrees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the final result. The max and min levels represent the turn of the player and the adversary, respectively.

Thus, mini-max search algorithms with alpha-beta cutoff optimizations are important programming techniques. Given below is another example of traditional tic-tac-toe game, which explains how the min-max search (with alpha-beta cutoffs) works.

Figure 2.5 shows the possible moves generated from a tic-tac-toe position where X has made three moves and O has made 2 moves; it is O’s turn to move. This is “level 0”. At level 0, O has four possible moves. How do we assign a fitness values to each of O’s possible moves at level 0? The basic min-max search algorithm provides a simple solution to this problem: for each possible move by O in level 1, make the move and store the resulting 4 board positions. Now, at level 1, it is X’s turn to move. How do we assign values to each of X’s possible three moves in Figure 1.8? Simple, we continue to search by making each of X’s possible moves and storing each possible board position for level 2. We keep recursively applying this algorithm until

we either reach a maximum search depth, or there is a win, loss, or draw detected in a generated move. We assume that there is a fitness function available that rates a given board position relative to either side. Note that the value of any board position for X if the negative of the value for O.



Figure 2.6 Alpha-beta pruning example

To make the search more efficient, we maintain values for alpha and beta for each search level.

Alpha and beta determine the best possible/worst possible move available at a given level. If we reach a situation, like the second position in level 2, where X has won, then we can immediately determine that O’s last move in level 1 that produced this position (of allowing X an instant win) is a low valued move for O (but a high valued move for X). This allows us to immediately “prune” the search tree by ignoring all other possible positions arising from the first O move in level 1. This alpha-beta cutoff (or tree pruning) procedure can save a large percentage of search time, especially if we can set the search order at each level with “probably best” moves considered first.

Alpha-beta search can be made even faster by considering only a narrow search window (generally determined by guesswork based on experience). This is known as aspiration search. In the extreme case, the search is performed with alpha and beta equal; a technique known as zero-window search, null-window search, or scout search. This is particularly useful for win/loss searches near the end of a game where the extra depth gained from the narrow window and a simple win/loss evaluation function may lead to a conclusive result. If an aspiration search fails, it is straightforward to detect whether it failed high (high edge of window was too low) or low (lower edge of window was too high). This gives information about what window values might be useful in a re-search of the position.

**2.6. Adding heuristics for improvements**

Further improvement can be achieved without sacrificing accuracy, by using ordering heuristics to search parts of the tree that are likely to force alpha-beta cutoffs early. For example, in chess, moves that take pieces may be examined before moves that do not, or moves that have scored highly in earlier passes through the game-tree analysis may be evaluated before others. Another common, and very cheap, heuristic is the killer heuristic, where the last move that caused a beta-cutoff at the same level in the tree search is always examined first. This idea can be generalized into a set of refutation tables.

Heuristics formula used is,

Heuristics = 9999\*3cp + 99\*2cp + 9\*1cp + 1\*0cp +10000\*3up + 100\*2up + 10\*1up + 1\*0up

Where #cp: no. of lines with # computer pieces along the same line; and

#up: no. of lines with # user pieces along the same line.

**2.7 Conceptual Design**

The game of 4x4x4 tic-tac-toe can be visualized as four 4x4 grids stacked vertically on top of one another. Providing the third dimension to the board allows much more possible moves and winning ways than those in a normal game. The four grids allow 64 possible moves and 76 different ways of winning the game. In case of the 3x3x3 board, an advanced user can always find a way to win by placement of a piece at the center most spot. This move, which eliminates almost half the winning possibilities, is not possible in 4x4x4 board merely because of the absence of a single center most spot. To calculate the best possible move at any given stage, the computer looks 3 or 4 moves ahead. Looking 3 moves ahead generates a possibility of roughly 262,000 board states. The more the computer thinks ahead, more is the possibility of generating a more intelligent computer move. As a result, looking 4 moves ahead generated 16 million states and looking 6 moves ahead will generate and search through roughly 64 million states. From this, it is clear that, as we look ahead by more moves, there is an exponential increase in the computer response time. Hence, the game has to be coded to look as far ahead as possible but to maintain a reasonable response time.

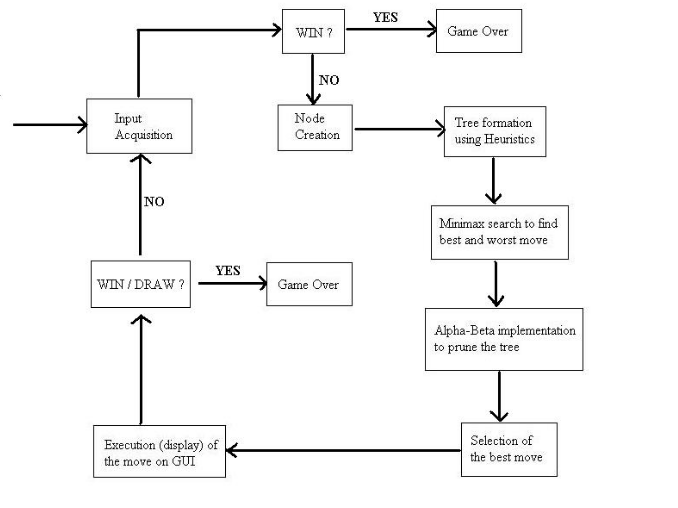
****

Figure 2.7 Block diagram 3D tic tac toe

Another point of consideration is determining the number of nodes for which information is to be stored in memory. As mentioned, a 4 ply game generates about 16 million nodes. Hence, these nodes are to be stored and accessed by recursion. In this way only 5 nodes are in the memory at any stage of implementation. This also enhances the use of alpha-beta pruning technique which is used to eliminate redundant nodes i.e. from which the possibility of winning is nil.

In figure 2.7, the block diagram illustrates the conceptual design.

As shown, the input is taken from user. A node is created and using heuristics tree formation takes place. Minimax search is applied to find the best and worst moves (nodes). Alpha-beta search is applied to eliminate unnecessary moves (nodes). Appropriate (best) move is selected and the move displayed on the board (GUI). Thus, a series of alternate user moves and computer moves are taken until a win is predicted for either of the sides.

**3. IMPLEMENTATION**

**3.1 System Requirements**

Desktop with following configuration

* **Operating system**

Tested on the following four operating systems

* + Windows 98
  + Windows 2000
  + Windows XP
  + Windows 7
* **Software**
  + JDK 1.5 or higher
  + Applet Viewer
* **Hardware**
  + Processor (Speed 1.6 GHz min)
  + RAM (96 MB min)

**3.2. Implementation techniques**

The implementation of the game is done in Java. The Graphical Interface is constructed with the help of Applets. Java Applets provide a simple and easy to use interface and is easy to create and maintain.

The implementation of the game is done as follows:

* Creation of game board on the GUI in Java applet providing a 3D perspective view of the game.
* Implementing mouse listeners in the code to determine the move played by the user.
* Creating and maintaining a tree like data structure which keeps a list of the nodes and possible future moves along with their heuristics value.
* Introducing Artificial Intelligence techniques to improve the game play by the computer.
* For each computer move played, the future of the board is calculated and analyzed systematically to determine the best possible move for both, the user and the computer.
* Tree nodes or possible moves with poor heuristics are avoided i.e. pruned from the tree structure. Moves with low heuristics are those which do not seem provide a winning possibility in the future.
* The best possible move thus calculated is shown on the User Interface. During the calculation of the move of the computer, mouse listeners are disabled so as to prevent the user from making successive moves.
* The game continues till either the user or the computer wins or all the possible positions are taken up by either of the players.

The GUI for the game is as shown in figure 3.1:

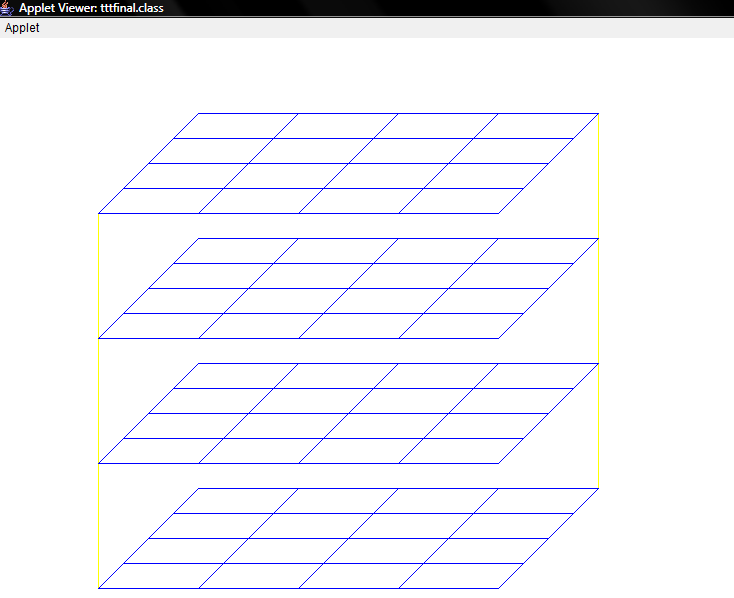


Figure 3.1 GUI

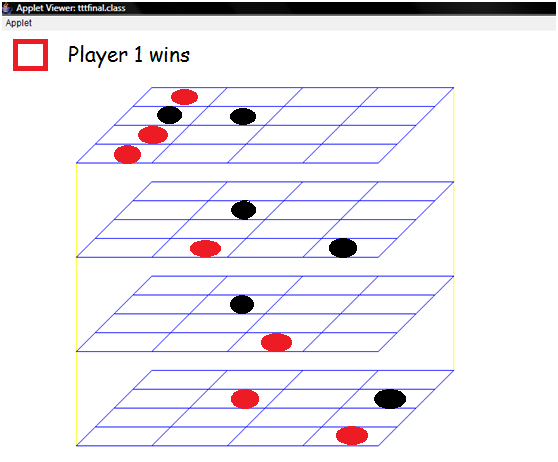
The various data structures used in the implementation of 3D tic tac toe:

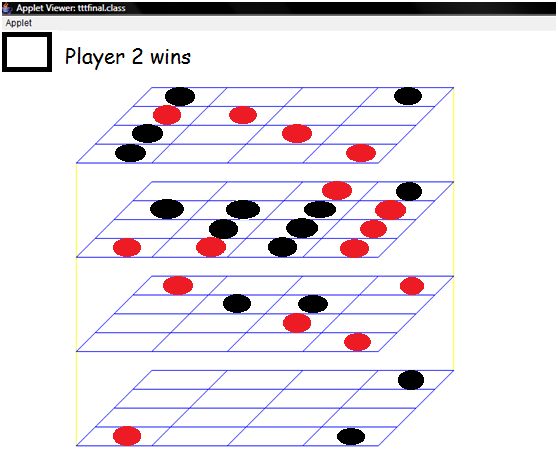
* Tree: These are basically the backbone of the implementation. All choices are first stored in the tree structure and then the various AI algorithms are applied to the trees. A tree structure is an algorithm for placing and locating files (called records or keys) in a database. The algorithm finds data by repeatedly making choices at decision points called nodes. A node can have as few as two branches (also called children), or as many as several dozen. The structure is straightforward, but in terms of the number of nodes and children, a tree can be gigantic. In a tree, records are stored in locations called leaves. This name derives from the fact that records always exist at end points; there is nothing beyond them. The starting point is called the root. The maximum number of children per node is called the order of the tree. The maximum number of access operations required to reach the desired record is called the depth. In some trees, the order is the same at every node and the depth is the same for every record. This type of structure is said to be balanced. Other trees have varying numbers of children per node, and different records might lie at different depths. In that case, the tree is said to have an unbalanced or asymmetrical structure.
* Linear Link list: These are used in the implementation to support for a BFS search on the tree. Link lists are basically variable length arrays. They can be used to store structured data. The link lists used in the implementation store the various leaf nodes of the tree at a particular instant.
* The table at each instant is stored in a 3D matrix of dimension 4x4x4.

The inputs to the program are taken from mouse clicks on the GUI. The output is shown graphically as an emulation of the 3D tic tac toe board on the screen using Applets.

**4. RESULTS**

**4.1 Results from the GUI**

****Figure 4.1 An image of user win

Figure 4.2 An image of computer win

**4.2 Discussions**

As seen from the results, the computer performs moves after deliberate processing and searching. The moves played by the computer are intelligent and challenging, giving the impression of playing with a human. The implementation program performs well, considering it’s a code that thinks ahead 4 plies. The alpha-beta pruning of the trees helps in better performance of the program.

**5. CONCLUSION**

**5.1 Conclusion**

Artificial Intelligence techniques such as minimax search, prediction of future of the board, alpha-beta pruning were studied and implemented in the game. Introducing heuristics further improved the game efficiency and makes the computer more intelligent.

**5.2 Future work**

Intelligent search techniques like alpha-beta pruning, minimax search and the use of heuristics in decision making, can be used in Game AI for achieving better results. The results of the report show that the game can be improved by implementing a 6 or 8-ply thinking code, by implementing better pruning and recursion techniques. The report can give insights into the use of these techniques in developing other intelligent algorithms.

**References**

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

[2] Artificial Intelligence: Implementing 3D Tic-Tac-Toe in C++, a paper by Bradley D. Bogenschutz, Department of Mathematics and Computer Science – Ripon College

[3] Intelligent Game Agents++, a paper by Bradley D. Bogenschutz, Department of Mathematics and Computer Science – Ripon College

[4] http://home.earthlink.net/~cmalumphy/3d.html

**Acknowledgement**

We take this opportunity to express our sincere thanks and gratitude to Dr. Gopakumaran Thampi, principal of Thadomal Shahani Engineering College, for providing the facilities and infrastructure for the project.

We further extend our thanks to Ms. Shalini Bhatia, for her constant guidance, support and critical review of the project.

We also thank the T.E Computer lab assistants for providing the necessary assistance throughout the experiments.

Lastly, we are grateful to Mr. Bradley D. Bogenschutz, author of the 3D Tic-Tac-Toe paper, for assisting us with the details of implementation before we embarked on this project.