**Automated Reasoning**

*Using propositional logic to represent various problems, and model checking and propositional inference to make conclusions*

**Daniel Saltz**

CSC 242 – Project 2

*Lab Partners: Sam Rusoff and Jack Gallagher*

1) Introduction

In order to represent complex problems, it is important to have a formal way of representing the state of the world. One way of doing this is through state-space-search, where the state of the world is represented

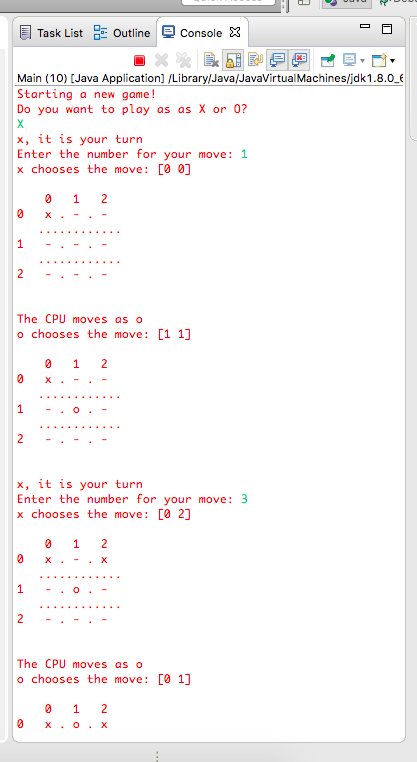
When developing a state-space search paradigm, there are many important factors to consider. This was our first challenge when creating a tic-tac-toe program through state-space search. The first task is to determine states within the state-space. The states are the arrangement of the tic-tac-toe board, where the X’s and O’s are located, and whose turn it is to move. The next part of state-space search to think about is the actions. Actions return the new state from executing a specific move. It is also important to define the initial and goal states. For tic-tac-toe, the initial state is the empty board, with X’s turn to move. Goal states, or terminal states, are situations where a player has three collinear X’s or O’s.

Given these definitions, we were able to proceed with the design of our project. Similar to many other state-space search problems, the algorithm took on a general form. We would continue to have the human alternate moves with the computer until a terminal state is reached. Each new move brought the algorithm from one state to another state through an action. While the user would be queried for a spot on the board, the computer would search for a move through the minimax algorithm.

2) Part 1 – Model Checking

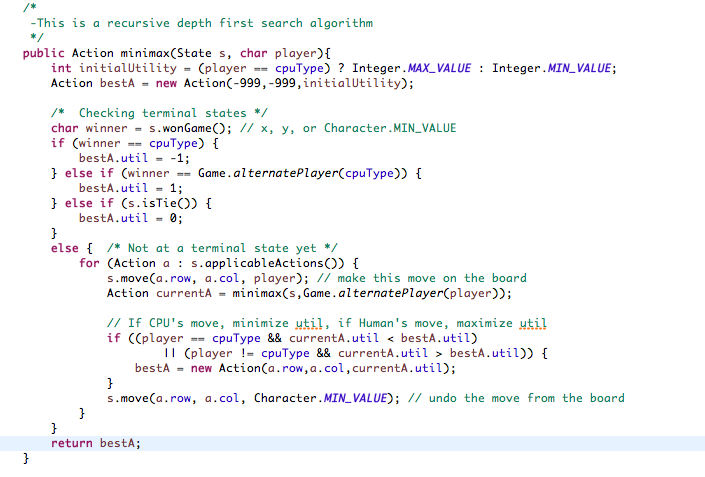
The first step in completing our single board implementation of Tic-Tac-Toe was to have the computer randomly select moves. This set up a framework for having a method return the computer’s move. Next, when we started to create more intelligent decisions for the computer, all we had to do was “plug” the new method into the chooseMove() method. Next, we wanted to implement minimax so that the computer would select the optimal move. In tic-tac-toe, an “optimal” move will never lead to defeat. It will lead to a draw if the opponent also plays optimally, and could lead to victory if the opponent makes a mistake.

We completed this algorithm and the computer cannot be beaten. Here is a screenshot of the computer properly blocking X’s attempt to win on the top row.



3) Part 2 – Inference Methods

It is worth analyzing minimax to determine its strengths and weaknesses. Minimax does as well as any algorithm can do for a single board tic-tac-toe game. That being said, it still has deficiencies. Minimax does not take into account the likelihood of “tricking” your opponent into making a mistake. For example, say that there is one move that leads to one path towards victory and two paths leading to defeat while another move leads to 2 paths towards victory and 1 path towards defeat. Minimax will value these two moves the same and will simply choose the move that it came across most recently, while you could make the argument that the second move is better since a mistake by the opponent is more likely to lead towards the computer’s victory. In addition, bare bones minimax always evaluate the entire tree. This is sufficient for single-board minimax since the tree is relatively small. However, for most problems, including nine-board minimax, this is inconceivable. There are several methods for reducing the tree. A common technique is through alpha beta pruning, where branches that are known to not affect the actual decision making process are *pruned* out of the tree.



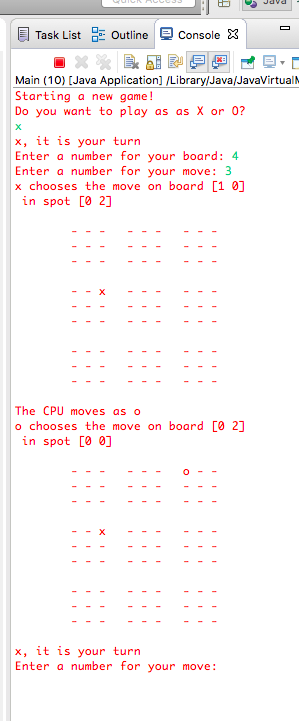
Recursive algorithms often provide a certain level of difficulty. They require a different perspective on the flow of the program, and it is often more difficult to think through the problem. Minimax is not a particularly difficult recursive algorithm, but it still took some tweaking to persist the data through the recursive calls properly. Figuring out how to use the information about terminal states to percolate that data up to the decision move was challenging. The key to solving this problem was to figure out the conditions under which the minimax() method returns. This occurs when a terminal state is reached, either by a victory by the computer, a victory by the human, or a tie. A -1, 1, or 0 is returned for these conditions, and eventually a value is returned all of the way up the call stack to reach the final returned value which contains the best utility, as well as a row and column for that move.

4) DPLL

At the beginning of the minimax method, the terminal states are checked in order to make sure that a terminal state has not been reached. If the current state is not a terminal state, the successor nodes are expanded to generate all of its successor states. The first successor is chosen, the move is made leading to a new state, and minimax is recursively called. This leads to the successor to be expanded into a new list, as long as it wasn’t a terminal state. This process continues until a terminal state is reached, in which case the utility is returned, and eventually the second element of the list is expanded. This searching strategy is depth first search because it expands the children nodes all the way down to the terminal states before expanding horizontally across siblings. In other words, a node’s children are expanded before its siblings. The amount of time the program will take to run can be thought of as the number of moves that the algorithm searches through. Since minimax is systematic, it searches every possible move. The algorithm searches down to node m, with a branching factor of b. Since b doubles with every new level of the tree, the time complexity can be thought of as (b, b2, b3, b4,…,bm), with a big O of bm. This algorithm takes big O of b\*m nodes of memory. This makes sense intuitively because every child that expands will have a branching factor of b, and there will be m children, for a totally of b\*m nodes.

5) Improvements

The nine by nine board Tic-Tac-Toe provides a different strategy than the single board game. Based on where the previous player played a move, that is the board that the next player must move on. This point becomes clearer with the screenshot below.



Because the user moved in the top right square of the middle left square (this would be the case for any board), the computer is required to make a move in the top right board. This changes the strategy of the game, because rather only searching through a 3x3 board, the search tree essentially expands to include nine boards. Due to this increased complexity, we were unable to run the simple minimax algorithm. The way that we implemented the nine-board data structure was through a four-dimensional character array. For the single board, we used a two-dimensional char array. It seemed most natural to create a two dimensional array of these single boards. This would allow us to logically search through each of the nine boards.

I analyzed many different techniques for being able to search for a move, in order to reduce the time and memory spent searching for a move. Iterative deepening combined with A\* is an algorithm that is able to use a heuristic to cutoff the search at a certain point and estimate a reasonable move. We decided to implement H-Minimax, which takes a similar approach. Rather than recursively run the minimax algorithm down to its terminal states, we would set a cutoff point. Once this number of iterations is reached, the algorithm would use a heuristic to make an estimate of the utility of that state. We were able to set this cutoff point and have the computer choose a random square in the appropriate board. However, we were unable to implement the heuristic.

5) Extra Credit

After learning about state space search algorithms more in depth, I feel that I am better equipped to write algorithms to solve complex problems. For this project, we were able to generate the optimal single-board Tic-Tac-Toe game, and successfully play nine board Tic-Tac-Toe using a cutoff and random moves in the appropriate boards. There are many interesting areas that I feel I can improve my algorithm. The first would be to implement a heuristic that is capable of very quickly estimating the utility of the current nine boards. This could be achieved by counting the number of moves away from winning (three collinear moves). Additional utility could also be given to a state if a fork is created – where there are two paths towards victory. Another way to improve the algorithm would be to reduce the number of branches in the search tree. This could be achieved by alpha/beta pruning or MTD(f).

It seemed intuitive to me that the first few moves of the nine-board game would be fairly simple. However, the minimax algorithm would spend it’s resources searching for a good move, where in reality, all the program needs to do is know a few basic rules to find a reasonably good move. I could combine the minimax algorithm would several basic logical if-else conditions to make optimal decisions in the first few moves, before always relying on the expensive algorithm.