Jacquelyn Moreno
CSC242 Project 1
NetId: jmore10

The goal of this project was to design, implement and evaluate an AI which plays Tic Tac Toe. There were two aspects, the first was to make a program that play basic Tic Tac Toe, and the second was to modify that program to play a more difficult variant of Tic Tac Toe. I began this process by defining the parameters of basic Tic Tac Toe as a state-space search problem. In state space search we must define 5 elements:

Set of states:

In tic tac toe, a state consists of two pieces of information, the configuration of X's and O's on the board, and the player whose turn it is next. The set of states is all possible configurations of these pieces of information.

Set of actions:

The set of actions is putting an 'X' or an 'O' in a position on the board.

Applicability:

In a given state, there are certain actions which may or may not be applicable. In Tic Tac Toe, an action is applicable in a state if both:

- The mark('X' or 'O') that is being placed on the board corresponds to the player whose turn it is.
- The position that the mark is being placed does not already have a mark in it.

Result:

The result of applying action A in state S is that S will now have the specified mark in the specified position, and it is the next players turn.

Cost:

The cost for each move is uniform.

For a specific instance of a state space search problem, we must also define:

Initial state:

The initial state of Tic Tac Toe is all positions of the board are empty, and it is X's turn to play.

Goal state:

The goal state in Tic Tac Toe is to get three consecutive positions in a row to contain the same mark.

In an adversarial search problem, we also define:

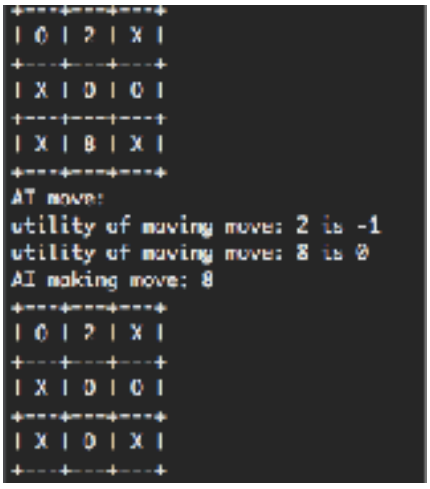
Utility function:

This function takes in a state and a player, and returns the utility of that state for that player. In Tic Tac Toe, this function will return 1 if a player has 3 in a row, 0 if the board is a tie, and -1 if the player has lost.

Terminal-Test:

This function takes in a state and returns true if the game is over, and false if it is not.

My next step was to implement these elements in code. I created a class to represent the board, and implemented various methods to transition between states, check applicability, check utility and check for a terminal state. After making all of the basic elements of Tic Tac Toe, I made a class which decides the AI's optimal move. In this class I implement the minimax algorithm, which returns the best move assuming that the opponent plays optimally.



Here is a sample of the minimax algorithm in action. The AI is mark 'O'. Given the first state with available moves 2 and 8, it considers each move in turn. If the AI makes move 2, then the opponent can make move 8

and win the game. Therefore, this move has utility -1. If the AI makes move 8, it blocks the opponents 3 in a row. Then, the opponents only remaining move is move 2 which results in a tie. Therefore, move 8 has utility 0. Since move 8 has greater utility than move 2, the AI makes move 8. The minimax algorithm allows the AI to play basic Tic Tac Toe, satisfying the first requirement.

The next part of this project was to implement 9-board Tic Tac Toe. A move in this variant of Tic Tac Toe is two numbers, the first corresponding to the board the current player wants to make a move on, and the second corresponding to the position in the specified board. The goal of 9-board is to win 1 board out of the nine, with the rule that the current player must make their move in the board corresponding to the position of the last players move. For example, if one player makes move (1 2), the next player must make their move on the 2nd board in any of the available positions. The new definitions for state space search are similar to the definitions for basic Tic Tac Toe. The set of states is the same except it is now all possible configurations of the 9-board. The set of actions is the same, and the applicability has the added rule stated above. The result and cost definitions are the same.

I created very similar classes and methods for my 9-board, with adjustments for the new parameters. The main changes were in my adversarial search method. I implemented minimax with alpha-beta pruning, and included a heuristic estimate. Alpha-beta pruning cuts down the number of states which need to be examined by keeping track of the best solution that the search has currently found, for both maximizing and minimizing players, and eliminating any states which are worse than the current best solution for max or min. This works because those states would never be reached anyways, assuming that both players make optimal decisions. After making this adjustment to my minimax, I noticed that the search was still taking upwards of two minutes to find the optimal solution. In an effort to cut down this runtime, I decided to add a heuristic evaluation function which is called after a certain depth. After a depth of 15, my program estimates the utility of the state based on the number of marks in a row

that are on the given board. If the current player has two marks in a row, the state has a utility of +1. If the opponent of the current player has two marks in a row, the state has a utility of -1. Otherwise, the utility of the given state is 0. The added depth limit and heuristic estimate cut down my runtime to drastically. If I were to add anything to my program it would be another heuristic to make my advanced Tic Tac Toe AI even smarter.