Kyle Edgette

CSC 242

Project 01 – Write Up

February 14, 2016

<u>I DO NOT WANT TO BE ENTERED IN THE COMPETITION</u>

Simple Tic-Tac-Toe

My Class design for simple tic-tac-toe consisted of Action, State, and Node. The Enum Player is also in the basic package. A brief description of the classes:

- **State:** A State contains a two dimensional array of Players and a Player field that is used to keep track of who is currently making their move.
- **Action:** An Action contains a Player field for the player who is currently making this action and an integer represent which space on the board the player wants to take. (Following the provided convention the accepted values are from 1 to 9).
- **Node:** A Node contains a State, an integer representing the utility value of that State, and an ArrayList of Nodes that are the successors of that State. The Node class also contains a static Player variable used to store whether the computer is X or O.

Stochastic Search

After I formalized this model, I implemented a simple, stochastic version of Professor Ferguson's "Coolest Program Ever" for a two-player game. In my version, the agent randomly chooses an applicable action and applies it to the state. Although simple and not "intelligent", this approach allowed me to solidify the mechanisms for getting a list (specifically an ArrayList) of applicable actions to the state and for applying an action to the state (i.e. updating the state).

Lastly, this initial approach allowed me to set up the gameplay "engine." Once the agent was able to "choose" a move and apply it, I designed the engine that asks the user for input, checks if their move is valid, applies their move, and then allows the agent to choose its move and apply, so on and so forth. At this point, I developed the methods to test if a State was a terminal State and, if it was, to determine who had won the game.

Adversarial Search

Now that the basic game was formalized and playable, I worked on improving the agent's decision making. My second version was an implementation of adversarial search, using the MiniMax algorithm. I developed the Node class at this point as a wrapper for the State class. The Node class adds the functionality needed to implement the MiniMax algorithm. In this implementation, after finding a State's successors, MiniMax was recursively called on each of the root's successors and the static field that kept track of the computer's Player is used to determine if the current "level" is a max or min level. Once the original State's utility (really, the Node's utility field) has been calculated, the agent iterates through the State's children and finds the child with the root's utility and that is the next State of the game. (The location that led to this new State is calculated after the fact and printed to Std.Out).

Note: It was at this point that I began forming the model for Super Tic-Tac-Toe, but I eventually returned to the basic version to implement the following changes.

Adversarial Search with Alpha-Beta Pruning

Next, I improved my MiniMax algorithm by implementing alpha-beta pruning, because the decision tree for SuperTTT would need to be pruned in order to search the State space adequately and efficiently. I used the outline for alpha-beta pruning from *Artificial Intelligence: A Modern Approach* (page 170, Third Edition) as a guide for implementing the algorithm. I did modify the "cutoff" points from what is described in the book, because I found that the agent was not pruning the tree using the cutoff in the book.

```
if( utility <= alpha){break;} CHANGED TO if(beta <= alpha) { break; }
```

Lines 99 to 104 in SuperNode

While this change seems trivial and not worth mentioning, a significant amount of time was dedicated to figuring out why the agent was not correctly pruning the tree, and this change fixed that problem.

Improvements

There is not much I would have changed algorithmically for the agent solving simpleTTT, except implementing a way for the agent to ignore rotations of the board that have already been previously analyzed. For example, if the user goes first and selects the middle location (i.e. spot 5) the agent only needs to consider 2 moves, a corner and a non-corner. All other choices are just rotations of those two.

I do wish my code was cleaner and a little more nicely designed, but I'm sure everyone wishes that.

Super Tic-Tac-Toe

My Class design for super tic-tac-toe consisted of SuperAction, SuperState, SimpleBoard, and SuperNode. A brief description of the classes:

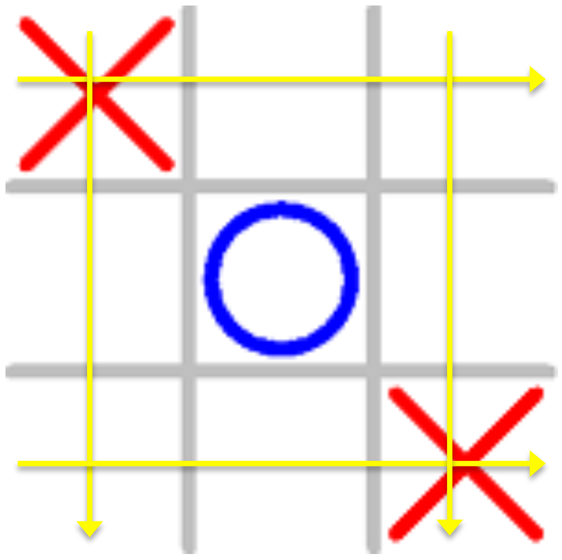
- **SuperState:** A SuperState contains a two dimensional array of SimpleBoards and a Player field that is used to keep track of who is currently making their move.
- **SuperAction:** A SuperAction contains a Player field for the player who is currently making this action, an integer represent which regular TTT board on the large board the player wants to play in, and an integer representing where on that board the Player wants to place their token. (Following the provided convention the accepted values are from 1 to 9 for both which board to play on and which location in that board to take).
- **SuperNode:** A SuperNode contains a SuperState, an integer representing the utility value of that State, and an ArrayList of SuperNodes that are the successors of that SuperState. The SuperNode class also contains a static Player variable used to store whether the computer is X or O.
- **SimpleBoard:** A SimpleBoard is "essentially" the same as the State from SimpleTTT, except a SimpleBoard does not have a utility, because the SuperState it is a member of has the utility. (*Note:* More on the difference between State and SimpleBoard classes in *Improvements*).

Alpha-Beta Pruning

After developing the alpha-beta pruning version of MiniMax discussed earlier and implementing it for SuperTTT, I realized that the tree was not adequately being pruned. At this point I looked at the Wikipedia page for alpha-beta pruning and implemented the pseudo-code found there. This new version did adequately prune the decision tree, but it was not making intelligent moves for some reason. I tried fixing this, but no matter what the agent would not make winning moves when given the chance and would not block me from winning if given the chance.

At this point I made the adjustment to my original pruning algorithm shown in the SimpleTTT section, and that algorithm pruned correctly and was able to evaluate up to a depth of 8 in the decision tree.

Lastly, I implemented a heuristic function for evaluating non-terminal states. It is a simple, but fast heuristic. Essentially, for a given board and a given Player, calculate the number of ways that Player can win. It is easier to code this by assuming the Player can win in all 8 ways on a TTT board, and then subtracting 1 point for each win condition that is blocked by the opposing player's token. In terms of Player X, the following arrows represent ways X can win on the given board:



The heuristic function would evaluate this state as a 4 for Player X and a 3 for Player O, in terms of their own utilities. Therefore the heuristic can return values from 8 to -8. This change meant the utility function for terminal States had to be updated to assign a value of 9 to winning states and -9 to losing states, to give them higher priority.

The last modification to the alpha-beta pruning algorithm was the implementation of a depth tracker that tracks the depth of the tree at any point. When the tree has gone as deep as specified, the State is evaluated by the heuristic function.

Improvements

Sadly, although my alpha-beta algorithm does traverse a significant portion of the tree, it does not always make the most intelligent decisions. Although it now takes winning moves when given the chance, it will still "send" me to a board where I can win and end the game. I'm not sure how I would fix this, but it is obviously the highest priority fix.

Lastly, again I wish my code were cleaner. I essentially unnecessarily redesigned a class (SimpleBoard) that already existed (basic.State) because I did not want the individual TTT boards in the SuperTTT board to have a utility field. This decision resulted in a lot of unnecessary/messy code, instead of using inheritance or another design.