CSC 242 Project 1

Ali Hashim

February 13, 2018

1 Introduction

In this project, a basic implementation of Tic-Tac-Toe was created and the AI player's move was chosen using the minimax algorithm with alpha-beta pruning. Additionally, an advanced variation of Tic-Tac-Toe was created called 9-board, consisting of 9 basic Tic-Tac-Toe boards. The game is over when one player wins in an individual board or every board is full. When one player plays in a position on a board, the opposing player has to play in the board corresponding to that played position. Nine-board expands the state-space significantly making the minimax algorithm slow and impractical. A heuristic was used to transform the minimax algorithm to an H-minimax. The depth to which the algorithm expands the tree is limited, and once the depth limit is reached, a heuristic is used to estimate the cost of the state, so that the algorithm doesn't have to search the whole state-space tree. Finally, ultimate Tic-Tac-Toe was implemented. Ultimate Tic-Tac-Toe is similar to nine-board, however the game doesn't end when an individual board is won; three child boards need to be won in a row, column, or diagonal like in basic Tic-Tac-Toe. The implementation required a tic-tac-toe board that was updated when an individual board in 9-board was won.

The definition of minimax is defined below:

$$
\begin{aligned}
&\textsc{Minimax}(s) = \\
&\textsc{Utility}(s) && if\ \textsc{Terminal-Test}(s) \\
&max_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s, a)) && if\ \textsc{Player}(s) = MAX \\
&min_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s, a)) && if\ \textsc{Player}(s) = MIN
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{H-Minimax}(s, d) = \\
&\textsc{Eval}(s) && if\ \textsc{Cutoff-Test}(s) \\
&max_{a \in \textsc{Actions}(s)} \textsc{H-Minimax}(\textsc{Result}(s, a), d + 1) && if\ \textsc{Player}(s) = MAX \\
&min_{a \in \textsc{Actions}(s)} \textsc{H-Minimax}(\textsc{Result}(s, a), d + 1) && if\ \textsc{Player}(s) = MIN
\end{aligned}
$$

The above definitions were used to create the adversarial searching algorithms. Minimax works by searching the entire state space tree until it reaches a goal state. The terminal state is evaluated at the leaves and is bumped up by choosing the branch that is best case for the max player in the worst case scenario, or when the min is playing optimally. H-Minimax is just a modified version of minimax that incorporates a depth cutoff, and a heuristic to estimate how good a state is.

2 State-Space Problem Definitions

Basic Tic-Tac-Toe

The formal state-space problem is defined below, and I elaborate how each element is integrated into my code:

- State: The arrangement of X's, O's, and blanks on the board, and which player's turn it is.

$$S = \begin{Bmatrix} M_{1,1} & M_{1,2} & \dots & M_{1,i} \\ M_{2,1} & M_{2,2} & \dots & M_{2,i} \\ \vdots & \vdots & \ddots & \vdots \\ M_{i,1} & M_{i,2} & \dots & M_{i,i} \end{Bmatrix}$$

  where $M \in X, O, \; or \; Blank$, i is the maximum length of the board (3 in this case), and $turn \in X \; or \; O$

- Actions: Placing a mark (X or O) on the 3x3 board. $A = \{1, 2, ...i^2\}$ where $\frac{A}{i}$ and $A \; \% \; i$ are the locations of the input in the row and column, respectively and M = turn

- Transition Model:

  - Applicability: Placing a mark in an empty spot on the board if it's that players respective turn.

$$\textsc{Actions}(s) = A \; if \; M_{A/i, \; A\%i} \; is \; Blank \; and \; getTurn = turn$$

  - Result: Updating the board to dispay the new mark in the board and change which player's turn it is.

$$\textsc{Result}(s, a) = M_{A/i, \; A\%i} = turn; \; turn = !turn$$

- Cost: 1 per move. c(s, a, s') = 1.

- Initial State: Empty board; X goes first

$$s = M_{A/i, \; A\%i} = Blank \; for \; all \; A; \; turn = X$$

- Terminal State: A board containing three of the same marks in a line either horizontally, vertically, or diagonally or a board that is full yielding a draw.

$$M_{r,1} = M_{r,2} = M_{r,3} \; || \; M_{1,c} = M_{2,c} = M_{3,c} \; || \; M_{1,1} = M_{1,2} = M_{1,3} \; || M_{1,3} = M_{2,2} = M_{3,1}$$

  where r is the row, c is the column and M cannot be blank.
  Draw terminal state = numMoves == 9

Advanced Tic-Tac-Toe

The formal state-space problem is defined below, and I elaborate how each element is integrated into my code:

- State: The arrangement of X's, O's, and blanks on the 3x3 board of basic tic-tac-toe boards, which player's turn it is, and the active board, or the board that the current player has to play in.

$$S = \begin{Bmatrix} B_{1,1} & B_{1,2} & \dots & B_{1,i} \\ B_{2,1} & B_{2,2} & \dots & B_{2,i} \\ \vdots & \vdots & \ddots & \vdots \\ B_{i,1} & B_{i,2} & \dots & B_{i,i} \end{Bmatrix}$$

  where B is defined as the state of the basic tic-tac-toe board defined earlier, i is the maximum length of the 9-board (3 in this case), $turn \in X\ or\ O$, and activeBoard = {null, 1, 2, ... 9} where null indicates a free play.

- Actions: Placing a mark (X or O) on the 3x3 board of boards. $A = < B, M >$ where B indicates the board number and M indicates the position on that board.

- Transition Model:

  - Applicability: Placing a mark in an empty spot on a particular board if it's that players respective turn and that board is active or there are no active boards.
    $\textsc{Actions}(s) = a\ if\ M_{A_M/i,\ A_M\%i}\ is\ Blank,\ and\ B_{A_B/i,\ A_B\%i} = activeBoard\ or\ activeBoard = null,\ and\ getTurn = turn$
  - Result: Updating the board to dispay the new mark in the 9-board, change which player's turn it is, and change what the active board is.
    $\textsc{Result}(s, a) = B_{A_B/i,\ A_B\%i}, M_{A_M/i,\ A_M\%i} = turn;\ turn = !turn; activeBoard = A_B$. if A_B isGameOver then activeBoard = null

Ultimate Tic-Tac-Toe

  - State: The arrangement of X's, O's, and blanks on the 9-board, which player's turn it is, the active board, and a global tic-tac-toe board whose state is dependent on which player wins in a child board in the 9-board.

    $S = < state(NineBoard),\ state(Board),\ turn = getTurn,\ activeBoard = getActiveBoard >$

  - Actions: Placing a mark (X or O) on the 9-board.

$$A = getActions(NineBoard)$$

  - Transition Model:

* Applicability: Placing a mark in an empty spot on a particular board if it's that players respective turn, that board is active or there are no active boards, and if that board is not in a terminal state.

  $\textsc{Actions}(s) = A\ if\ Board_{A_B}\ is\ Blank\ and\ activeBoard = A_B\ or\ activeBoard\ is\ null$

* Result: Updating the board to display the new mark in the 9-board, change which player's turn it is, change what the active board is, and updating the global board if a child-board in the 9-board reaches a terminal state.
  $\textsc{Result}(s, a) = \textsc{Result}(state(NineBoard), a)$, if B in NineBoard is inTerminal-State, $Board(M_B) = getWinner(B)$

– Initial State: Empty board; X goes first; There is no active board.
  I = initialState(NineBoard) and initialState(Board), turn = X.

– Terminal State: A global board containing three of the same marks in a line either horizontally, vertically, or diagonally or a board that is full yielding a draw. Those marks are updated when a child board in the 9-board reaches a terminal state.
  T = if Board is inTerminalState

- Cost: 1 per move. c(s, a, s') = 1

- Initial State: Empty board; X goes first; There is no active board.
  I = I for each child board in 9-board

- Terminal State: A 9-board that consists of one board in a winning state which yields a win for that player or a 9-board that consists of every board in the draw terminal state which yields a draw.
  T = any board B in the 9-board inTerminalState with board not drawn.

3 Implementation

The following code snippet shows the interface that I used for each game. Board, NineBoard, and UltimateTTT all implement the State interface. I decide to implement an interface for each of the versions of Tic-Tac-Toe instead of subclassing the basic Tic-Tac-Toe board because it makes clear what functions are necessary to understand the state-space problem. Subclassing might've reduced the amount of redundant code, but it is easier to read the code this way.

```
public interface State {
        Mark getTurn();
        void changeTurn();
        boolean isGameOver();
        boolean move(int board, int input, Mark player);
        void printBoard();
        Mark getWinner();
        ArrayList<Move> getPossibleMoves();
}
```

The structure of the code base for each game goes as follows: A board class, a user input class, and a adversarial search class. The board class keeps track of the state of the board, the user input class gets the user's input, and the adversarial search class gets the best move for the AI. The NineBoard class is just a 2D array of Boards, with an additional tracker of the active board. The UltimateTTT class consists of a nine board and a regular board that keeps track of the children winners of the nine board.

To return the move decided by minimax, I modified the code from the pseudo-code explained in the book. Since getMax() or getMin() only returns the scores and not the actual moves, I implemented the getAIMove() to make a move. The getAIMove() method is redundant to getMax(), but keeps track of the best move:

```
public Move getAIMove(){
        int maxScore = -Integer.MAX_VALUE;
        Move move = null;
        long start = System.currentTimeMillis();
        for (Move test: board.getPossibleMoves()){
                Board copy = board.deepClone();
                copy.move( test.getBoard(), test.getPosition(), bot
                int score = minimaxWithAlphaBeta(copy, -Integer.MAX
                if (score > maxScore){
                        move = test;
                        maxScore = score;
                }
        }


}
```

Heuristic Implementation

The heuristic that I implemented estimated how good a state was via the rules below:

- The more marks that one player has in a row, column, or diagonal in which the opposing player has no marks, the better for that player and equally bad for the opposing player. The sum of the scores in each board and in each row, column, and diagonal would be summed up and added to the total score.
- The more free plays that one player has in a board, the better for that player and equally bad for the opposing player. The number of free plays for each board is added to the total score.
- The heuristic is weighted as such:
  - Add $2^{botCount}$ and subtract $2^{userCount}$ where botCount is the number of marks the max player has in a row, column, or diagonal in which the opposing player has no marks in and userCount is just the same but for the min player. If there is a single board with 2 X's in a row with an O in the middle position then the heuristic score for that state is

$2^2 + 2^1 - 2^1 - 2^1 = 2$. The X player has two in a row and 1 in a column where the O player has 1 in a diagonal and 1 in a row.

- Add 2 times the number of free plays for the max player in a given state for each board and subtract 2 times the number of free plays for the min player.
- The score of each board is added to the heuristic score of the entire state
- For Ultimate Tic-Tac-Toe, the state of the global board, or the board that keeps track of the winners in the nine board, is weighted 100 times more to the heuristic than each child board in the nine board because the global board decides who wins the game.

The cutoff depth for H-minimax was decided to be set to 5 for both advanced tic-tac-toe and for ultimate tic-tac-toe. This cut-off was decided based on setting the depth to the largest value where the runtime was limited to 1s for a play on a certain board. This runtime was measured for when the active board was set, and it wasn't a free play. An improvement would be to lower the depth cutoff for the free plays because that takes much longer to run.

Another overhead cost of the algorithms is that the state has to be copied before moves are tested to get the best move. I implemented the copy by implementing the serializable interface. Copying resulted in the biggest challenge for me in this project as just copying it with a constructor still modified the state via reference type instead of value type.

4 Analysis

Basic Tic-Tac-Toe is an ideal environment to test adversarial search. Tic-Tac-Toe is a fully observable, deterministic, zero-sum game. The maximum state-space search tree occurs when there are initially no marks on the board resulting in 9! or 362880 nodes. The time complexity of minimax is $O(b^m)$ and the space complexity is $O(bm)$

Alpha-Beta Pruning Analysis

The advantage of implementing alpha-beta pruning is that it limits the number of branches visited. On the random case, it reduces the runtime of minimax from $O(b^m)$ to $O(b^{\frac{3m}{4}})$. The pruning works by keeping track of the lowest bound for the max player and the upper bound for the min player. If the value is worse than alpha for a max node or worse than beta for a min node, then the successors of those nodes don't need to be visited.

To test if my implementation of Alpha-Beta Pruning worked, I kept track of how many nodes were visited when it was implemented and when it wasn't implemented, and I also tested the worst case runtime (when there are no marks on the board).

	Runtime (ms)	**Nodes visited**
With α, β pruning	18	30709
Without α, β pruning	125	549945

Table 1: Table

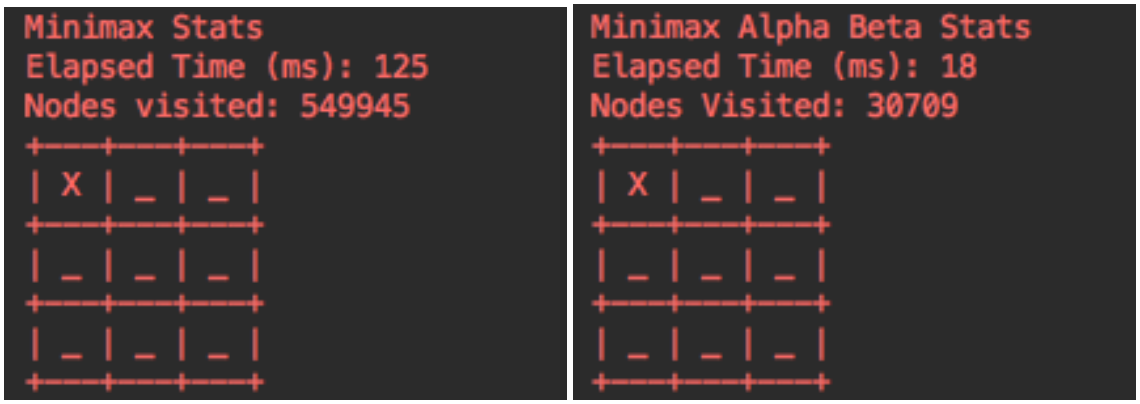

Figure 1: Screenshots depicting minimax stats

These figures show that alpha beta pruning significantly lowers the number of nodes visited and the runtime. The nodes visited is roughly 20 times lower than without alpha beta pruning, and the runtime is about 10 times faster.

Nine Board Analysis

The 9-Board variation of tic-tac-toe is interesting because it uses the same principles of basic tic-tac-toe, adds a layer of complexity, and significantly changes how the game is played strategically. The maximum number of moves in a game is now 9 times as large as in regular tic-tac-toe, and the game is much less likely to end in a draw. 9-board increases the maximum state space tree from 9! to 81! nodes. This renders regular minimax impractical. The minimax algorithm remains generically the same with the addition of a heuristic to estimate how good a state is at a certain depth cutoff, so that the algorithm doesn't search the entire tree.

The following images show some initial moves in the 9 board:

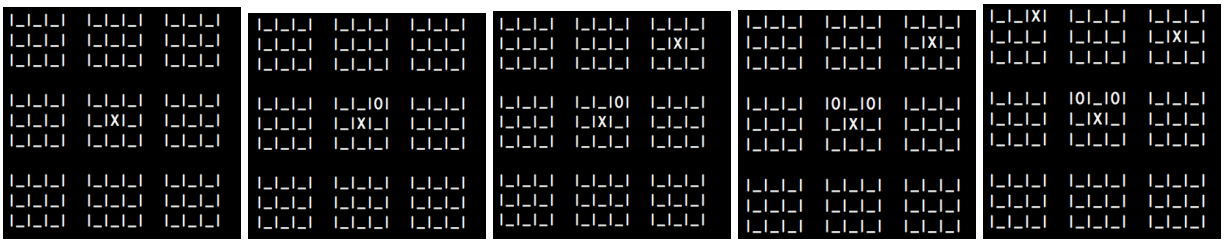

Figure 2: Figure depicting initial moves in a nine board game. The AI is X, the user is O

The heuristic enables the AI to play in a good spot to win. The first move the AI makes is to play in the 5th position of the 5th board, which is the best place to start because there are more winning states. After the user plays in a different position, the AI still plays in the 5th position. This puts the AI in an immediate disadvantage, however the later turns will avoid the 5th board.

Heuristic Evaluation

Advantages of the heuristic

- The heuristic maximizes the number of possible winning states. Because it incentivizes rows, columns, and diagonals with more marks for the max player with no marks for the min player, the heuristic naturally scores positions that have more possible winning spots. For example, the first move the AI makes is usually the middle position on a board.
- The heuristic doesn't make a stupid move. This might not be the best move, but the AI won't play in a position that gives the opposing player an immediate advantage. The cases in which the AI loses is when it has no best choice (i.e. no matter what position it plays in the opponent can win in that corresponding board).
- The heuristic takes into account the huge advantage of free plays. When a player can choose the board to play in, they oftentimes win. This is more significant in Ultimate Tic-Tac-Toe as free plays occur more often than in 9-board, but the AI is unlikely to play in a position in which the corresponding board has a draw.

Disadvantages of the heuristic

- The heuristic plays defensively. It only takes into account the states of individual boards and sums them together. The heuristic could be improved if it could trap the opponent into playing into a board that is disadvantageous to them by providing a heuristic that estimated how good a state is by looking at the dependency of the boards.
- The heuristic does not estimate the state well in Ultimate Tic-Tac-Toe. When the game becomes more complex, the heuristic becomes less intuitive to implement. Because the heuristic doesn't account for the dependency of individual boards, it is hard to truly evaluate the state well. This becomes a problem with Ultimate TTT because the dependency of the boards is crucial.

UltimateTTT Analysis

Ultimate Tic-Tac-Toe is a small variation from 9-board. A regular tic-tac-toe board keeps track of which child boards in 9-board are won, and the game is over if that global board ends in a terminal state. This does not impact the size of the space tree significantly, but does effect how good the heuristic is.

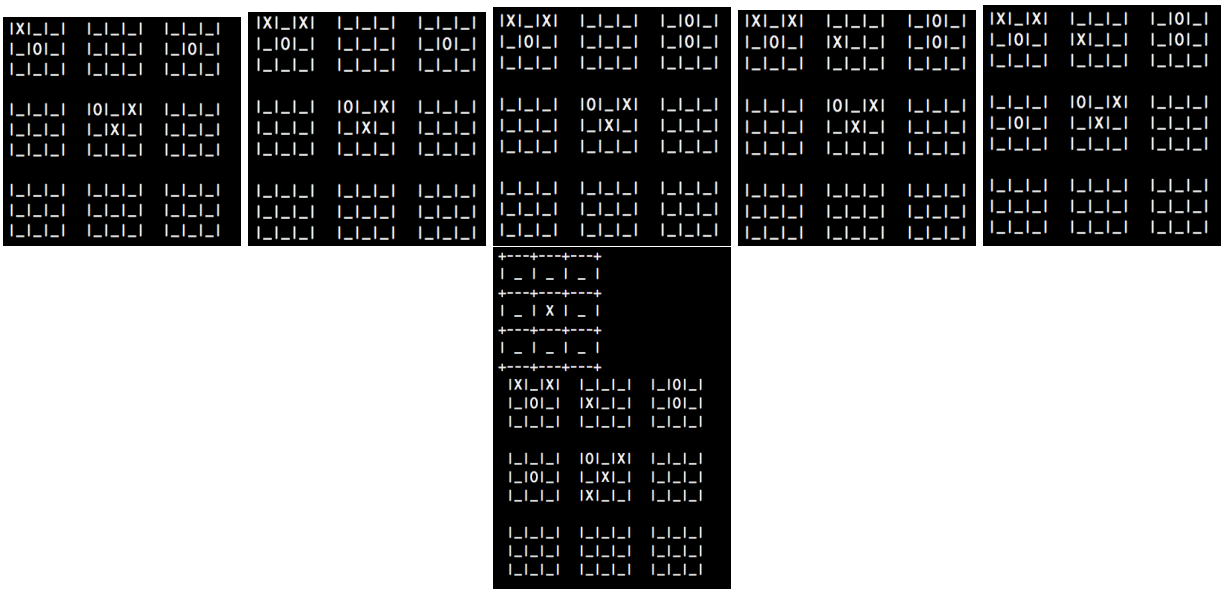

Figure 3: Figure depicting moves in a ultimate TTT game. The AI is O, the user is X

From the figure above, the AI plays in the 5th position frequently like it initially did in nine board. The ultimate TTT version continues to play in the 5th position until the user wins the 5th board. This figure shows how defensive the algorithm is because it takes a couple of bad moves before the algorithm plays well. After the 5th board is taken, the algorithm chooses the 5th position rarely because free plays for the user are weighted badly for the AI. This allows the AI to recover. If there was more time, I'd have configured the heuristic so that it does not play in the same position as frequently, so that it can spread the moves out and play offensively.

References

[1] Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.