CSC242 Project1 Write-Up
Chenwei Wu cwu59
Collaborators: Yi Yao yyao20, Haosong Rao hrao

Introduction

In this write-up I am going to analyze the state-space model analysis of the game, algorithms like mini max and alpha beta pruning, and the depth-limited search. I will also talk about how I got the heuristic values. For small problems like tic tac toe the whole tree search was enough to search through all the states and find the optimal solution however for Ultimate tic tac toe the possible states are just too many. I used alpha beta pruning and depth limited search to solve this problem since it does not require as much memory. In the end I also talked about what I have learned from this project and what can be possibly improved.

3X3

State-Space Model Analysis

In order to finish this project, I abstracted the tic tac toe game into a state space search problem.

Initial State: An empty board represented by a two-dimensional char array which is 3 by 3. Each position is represented by ‘ ’.(Empty)

Actions: Each move is stored in the State class, and through generating the next states, the actions are represented by either replacing the ‘ ’ (Empty) with ‘X’ and ‘O’.

Goal State: If the AI plays X, the goal state will be three continuous X in row or column or diagonals, which means it has won the game, or coming to a draw.(All positions filled up without a winner)

Transitional Model: The X or O is put into a empty position, generating a new state.

Path-Cost: Each step’s cost is equal and is 1. But in this game it doesn’t really matter.

Data Structures:

Because I am using the Minimax algorithm, which I will discuss soon, I am taking use of the tree search. Since you just can not fill an position that already has a ‘X’ or ‘O’ marked on it, the worrying of loopy structure meaningless.

Building up the basic frame of the boards was not a harsh work. As described in the state-space model analysis, I used a state class to store the current positions of the board using the 2d array, a turn counter to decide whose turn it is, two ints called actionrow and actioncol to record the actions, a char winner to get the result:

```
public class State {

    char[][] board={
            {' ',' ',' '},
            {' ',' ',' '},
            {' ',' ',' '}
            };
    int actionrow=0;
    int actioncol=0;
    char winner='N';//initializing the winner to 'None'
    boolean HumanPlayerX = false;
    int turn = 0;
```

Also, I implement methods called temp(), to get a deep copy of the current board, so that when I try out the different moves in my tree, I don't have to risk changing my original board positions and can do it by just using the deep copy.

```
    //making a deep copy of the state so that we don't have to undo the actions
    //referring from http://javatechniques.com/blog/faster-deep-copies-of-java-objects/

    protected State temp() {
        char[][] temp = new char[this.board.length][this.board.length];
        for(int i = 0; i < 3 ;i++)
        {
            temp[i] = this.board[i].clone();
        }
        int turn2 = this.turn;

        State copy = new State(temp,turn2);
        copy.actioncol=this.actioncol;
        copy.actionrow=this.actionrow;
        return copy;
    }
```

Finally and most importantly, the AI decisions are made on the algorithm called minimax algorithm. It analyzes the terminal board states and return the value of winning or losing. As this is a two-players' zero-sum game, what is good for game pro A is exactly the same degree bad for game pro B. Then that means every time you reach a terminal state that has three continuous X that is a win for you and when you reach a terminal state that has three continuous O that means a loss for you. Being optimal, players are marked as maximizers and minimizers. By the following algorithm, we can always found out the most optimal solution to the current state and make the corresponding moves.

```
    //minimax recursive function
    public int evaluation(State board,int depth , boolean maximizer)
    {
        //taking the depth and final result of the boards into
consideration and evaluates
        if(board.checkWin()=='O')
        {
            return -10+depth;
        }
        else if(board.checkWin()=='X')
        {
```

```
            return 10-depth;
        }
        else if(depth > 8||board.checkWin()=='D')
        {
            return 0;
        }
        //setting different initial values for best depending on maximizer
or minimizer
        int best= (maximizer)? Integer.MIN_VALUE:Integer.MAX_VALUE;
        //generates the children of a certain node in the search tree
        ArrayList<State> allmoves = board.PossibleMoves();
        for(State i: allmoves)
        {
            //maximizer
            if(maximizer)
            {
            int test = evaluation(i , i.turn, false);
            best = Math.max(test,best);
            }
            //minimizer
            else
            {
                int test =evaluation(i,i.turn, true);
                best = Math.min(test,best);
            }
        }
        return best;
    }
```

Then, by simply going through all the possible moves that a current state have and comparing their values, we can easily find the optimal move.

In this case, with only factorial nine number of states, the time complexity is not a big issue. But for the 9X9 game, we have to use depth limited search and alpha beta pruning.

9X9

State-Space Model Analysis

Initial State: An empty board represented by a two-dimensional state array which is 3 by 3. So the 3x3 state(three-by-three boards) array combines to be a 9x9 board.

Actions: Each move is stored in the State class and Staten class, and through generating the next states, the changes in small boards are represented by either replacing the ‘ ’ (Empty) with ‘X’ and ‘O’ and the change in large board are represented by two ints called actionx and actiony, that stores the position of the next board.

Goal State: If the AI plays X, the goal state will be three continuous X in row or column or diagonals, which means it has won the game, or coming to a draw.(All positions filled up without a winner)

Transitional Model: The X or O is put into a empty position, generating a new state. However, different from the 3x3 one, the positions X and O are put will affect the next board's position.

Path-Cost: Each step's cost is equal and is 1. But in this game it doesn't really matter.

Heuristics:

I think the alphabeta pruning approach will work better with heuristic values. So I used the for loops to count all the two-in-a-row (row/column/diagonal) and transformed them into values. So this would help the AI learn better about what will lead to not only victory of game but also a prevailing attitude on the whole board.

Alpha-Beta Pruning Approaches And Depth-Limiting Search:

To improve program by reducing useless search, I took advantage of the alpha beta pruning.

We can think of the values this way:

a<=N<=b

Alpha is the lower bound while Beta the upper, if a>b there wont be N as a solution.

The algorithms works like this: The initial Alpha is -inf and beta +inf(Making both players start with worst value) When choosing a branch of a state, the min score that Minimizer is assured of is less than the Maximizer is assured of. (Alpha>=Beta) And in this way we can just skip the whole branches because the current state will not choose it! Here is the code:

```
if(is)
{
    for(Staten s: children)
    {
        bestVal =  alphabeta(bestMove,depth,alpha,beta,false);
        Staten choice = s.Temp();

        int tempmax = alphabeta(choice,depth,alpha,beta,false);
        if(tempmax > bestVal)
        {
            bestMove = choice;
            bestVal = tempmax;
        }
    }
}
else
{
    for(Staten s: children)
    {
        bestVal =  alphabeta(bestMove,depth,alpha,beta,true);
        Staten choice= s.Temp();
        int tempmax = alphabeta(choice,depth,alpha,beta,true);
        if(tempmax < bestVal)
        {
            bestMove = choice;
            bestVal = tempmax;
```

Also, this is not enough. Alpha Beta algorithm grew exponentially as the depth increased. I set the depth to 8 and the program is taking a few seconds to get the answer. By doing depth limited search we can do the programming so fast and when I set the depth to 6 the programs runs so much faster.

Learning and Possible Improvements:

This game really makes me learn a lot. First of all I am just so impressed by how much the code can do! I personally is a very bad ultimate tictactoe player , always defeated by others in few moves. But with the program I can at least make draws and even win sometimes. This shows the great potential of AI, what if it is not just tic tac toe but some more complex games, like my favorite, StarCraft? That would be so much more interesting and complicated.

Also, I could have possibly improved the program by working on the heuristic functions better. I could not think of better heuristic values methods so I used my current one. Also I could have shorten my code and running time by not using so many deep copies.