Hw-4

Sunday, September 27, 2020 9:30 PM

CSE3521_HW4

Authors: Lauren Saggar and Michael Seidle

1 c) The game does play as I would expect. The CPU player uses the move expand order specified by an array where it chooses the next consecutive available spot numbered 0-8 for each of its turns based on the minimax algorithm. Since this is no an optimal ordering, I can beat it. As expected, it is easier to beat the CPU player when I go first.

CPU move #1: (Evaluated 59705 states, Expanded 31973 states)
CPU move #2: (Evaluated 1055 states, Expanded 582 states)
CPU move #3: (Evaluated 27 states, Expanded 12 states)

Over the course of the game, the number of states evaluated decreases. This is to be expected, as every time that the CPU player takes a turn, the minimax algorithm is evaluated at a depth two lower than it's previous evaluation. This makes sense, as there are always going to be less available paths to a terminal state as the minimax algorithm is applied lower in the state tree.

e)

The game does play I as expect. The expanded shrinks by a good amount from standard minimax. Also good moves are made to prevent the human from winning. Not always the best best move but for easy choices to prevent wins it works as expected.

CPU move #1: (Evaluated 55505 states, Expanded 29633 states)
CPU move #2: (Evaluated 959 states, Expanded 510 states)
CPU move #3: (Evaluated 35 states, Expanded 17 states)
CPU move #4: (Evaluated 4 states, Expanded 2 states)

f)

Nodes expanded with Minimax: 294778 states
Nodes expanded with Minimax + Alpha-Beta Pruning: 31974 states

How do the two algorithms compare?

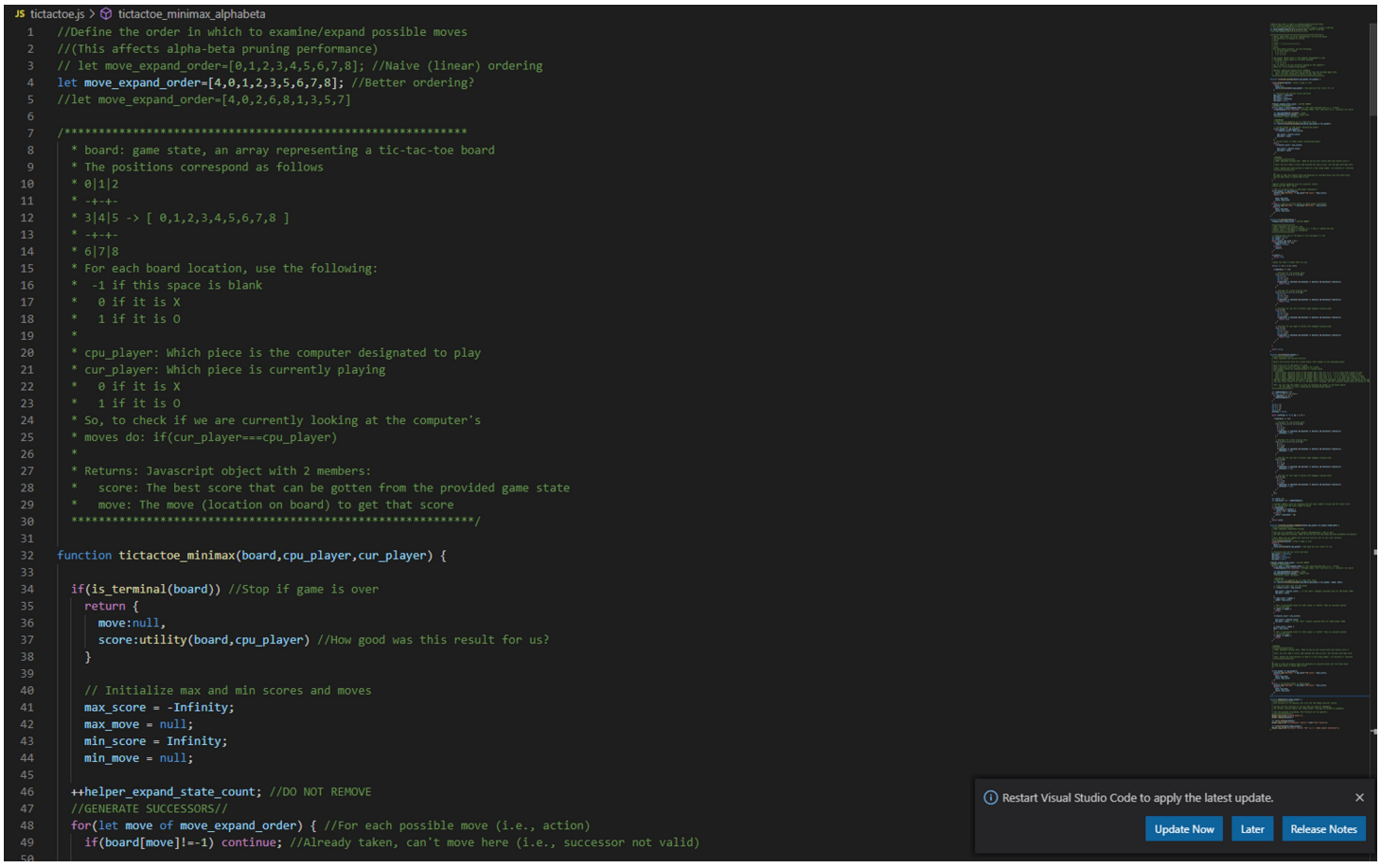
The number of expanded states dramatically decreased with alpha-beta pruning as expected. This is because alpha-beta pruning allows us to eliminate branches of the search tree that cannot possibly affect the final minimax decision without checking their states to improve searching efficiency.

Results with new move_expand_order:

Nodes expanded with Minimax: expanded 294778 states
Nodes expanded with Minimax + Alpha-Beta Pruning: expanded 29634 states

How do the results change? Why do you get this result?

The number of states expanded for alpha-beta pruning decreased with the new move_expand_order configuration as expected, as those states that are more likely to give a desired result are checked first when expanding new states. This in turn improves searching efficiency.



```
JS tictactoe.js > tictactoe_minimax_alphabeta
48    for(let move of move_expand_order) { //For each possible move (i.e., action)
49      if(board[move]!=-1) continue; //Already taken, can't move here (i.e., successor not valid)
50
51      let new_board=board.slice(0); //Copy
52      new_board[move]=cur_player; //Apply move
53      //Successor state: new_board
54
55      //RECURSION
56      // What will my opponent do if I make this move?
57      let results=tictactoe_minimax(new_board,cpu_player,1-cur_player);
58
59      // Current player is CPU player (maximizing player)
60      if(cur_player == cpu_player){
61        if (results.score > max_score){
62
63          max_score = results.score;
64          max_move = move;
65        }
66      }
67      // Current player is human player (minimizing player)
68      else {
69        if(results.score < min_score){
70
71          min_score = results.score;
72          min_move = move;
73        }
74      }
75
76      //MINIMAX
77      /**********************
78      * TASK: Implement minimax here. (What do you do with results.move and results.score ?)
79      *
80      * Hint: You will need a little code outside the loop as well, but the main work goes here.
81      *
82      * Hint: Should you find yourself in need of a very large number, try Infinity or -Infinity
83      **********************/
84
85      /*
86      We need to take the orginal board and generate all possible moves and from those moves
87      see how many moves it would take to win
88      */
89    }
90
91    //Return results gathered from all sucessors (moves).
92    //Which was the "best" move?
93
94    // Runs if current player is CPU player (maximizer)
95    if(cur_player == cpu_player){
96      console.log("CPU Move: "+ max_move+"CPU Score: "+max_score);
97      return {
```

Restart Visual Studio Code to apply the latest update.

Update Now | Later | Release Notes

```
JS tictactoe.js > tictactoe_minimax_alphabeta
96      console.log("CPU Move: "+ max_move+"CPU Score: "+max_score);
97      return {
98
99        move: max_move,
100       score: max_score
101     };
102   } else { // Runs if current player is human player (minimizer)
103     console.log("CRU Move: "+ min_move+"CRU Score: "+min_score);
104    return {
105       move: min_move,
106       score: min_score
107     };
108   }
109 }
110
111 function is_terminal(board) {
112   ++helper_eval_state_count; //DO NOT REMOVE
113
114   /************************
115   * TASK: Implement the terminal test
116   * Return true if the game is finished (i.e, a draw or someone has won)
117   * Return false if the game is incomplete
118   ************************/
119
120   // Checking base case if the board is full and game is a tie
121   let isFull = true;
122   let count = 0;
123   while (isFull && count < 9) {
124     if(board[count] == -1){
125       isFull = false;
126     } else {
127       count++;
128     }
129   }
130
131   if(isFull) {
132     return true;
133   }
134
135   //Check the other 9 states that are wins
136
137   for(var j = 0; j <= 6; j++){
138
139     if(board[j] != -1){
140
141       // Searches for row winning state
142       if(j == 0 || j == 3 || j == 6){
143         let a = j;
144         let b = j +1;
```

Restart Visual Studio Code to apply the latest update.

Update Now | Later | Release Notes

JS tictactoe.js > tictactoe_minimax_alphabeta

```
            let a = j;
            let b = j +1;
            let c = j+2;
            if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
              return true;
            }
          }

          // Searches for column winning state
          if(j == 0 || j == 1 || j == 2){
            let a = j;
            let b = j +3;
            let c = j+6;
            if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
              return true;
            }
          }

          // Searches for top left to bottom right diagonal winning state
          if(j == 0){
            let a = j;
            let b = j +4;
            let c = j+8;
            if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
              return true;
            }
          }

          // Searches for top right to bottom left diagonal winning state
          if(j == 2){
            let a = j;
            let b = j +2;
            let c = j+4;
            if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
              return true;
            }
          }
        }

      }

      return false;
    }

    function utility(board,player) {
      /**********************
      * TASK: Implement the utility function
      *
      * Return the utility score for a given board, with respect to the indicated player
      *
```

Restart Visual Studio Code to apply the latest update.

Update Now | Later | Release Notes

JS tictactoe.js > tictactoe_minimax_alphabeta

```
      * Return the utility score for a given board, with respect to the indicated player
      *
      * Give score of 0 if the board is a draw
      * Give a positive score for wins, negative for losses.
      * Give larger scores for winning quickly or losing slowly
      * For example:
      *   Give a large, positive score if the player had a fast win (i.e., 5 if it only took 5 moves to win)
      *   Give a small, positive score if the player had a slow win (i.e., 1 if it took all 9 moves to win)
      *   Give a small, negative score if the player had a slow loss (i.e., -1 if it took all 9 moves to lose)
      *   Give a large, negative score if the player had a fast loss (i.e., -5 if it only took 5 moves to lose)
      * (DO NOT simply hard code the above 4 values, other scores are possible. Calculate the score based on the above pattern.)
      * (You may return either 0 or null if the game isn't finished, but this function should never be called in that case anyways.)
      *
      * Hint: You can find the number of turns by counting the number of non-blank spaces
      *       (Or the number of turns remaining by counting blank spaces.)
      **********************/

      var numBlankSpaces = 0;
      for (var i = 0; i < 9; i++) {
        if (board[i] == -1) {
          numBlankSpaces++;
        }
      }

      let a = -1;
      let b = -1;
      let c = -1;
      let j = 0;
      setFound = false;

      while (setFound == false && j <= 6) {

        if(board[j] != -1){

          // Searches for row winning state
          if(j == 0 || j == 3 || j == 6){
            a = j;
            b = j +1;
            c = j+2;
            if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
              setFound = true;
            }
          }

          // Searches for column winning state
          if(j == 0 || j == 1 || j == 2){
            a = j;
            b = j +3;
            c = j+6;
```

Restart Visual Studio Code to apply the latest update.

Update Now | Later | Release Notes

```
JS tictactoe.js > tictactoe_minimax_alphabeta
238             b = j +3;
239             c = j+6;
240             if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
241               setFound = true;
242             }
243           }
244
245           // Searches for top left to bottom right diagonal winning state
246           if(j == 0){
247             a = j;
248             b = j +4;
249             c = j+8;
250             if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
251               setFound = true;
252             }
253           }
254
255           // Searches for top right to bottom left diagonal winning state
256           if(j == 2){
257             a = j;
258             b = j +2;
259             c = j+4;
260             if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
261               setFound = true;
262             }
263           }
264         }
265         j++;
266       }
267
268       let score = 0;
269       let num_moves = 9 - numBlankSpaces;
270
271       // Assigns highest score for winning with the least number of moves and the lowest score
272       // for losing with the least number of moves
273       if (setFound) {
274         if (board[a] == player) {
275           score = 10 - num_moves;
276         } else {
277           score = num_moves - 10;
278         }
279       }
280       return score;
281     }
282
283     function tictactoe_minimax_alphabeta(board,cpu_player,cur_player,alpha,beta) {
284       /**********************
285       * TASK: Implement Alpha-Beta Pruning
286       *
287       * Once you are confident in your minimax implementation, copy it here
```

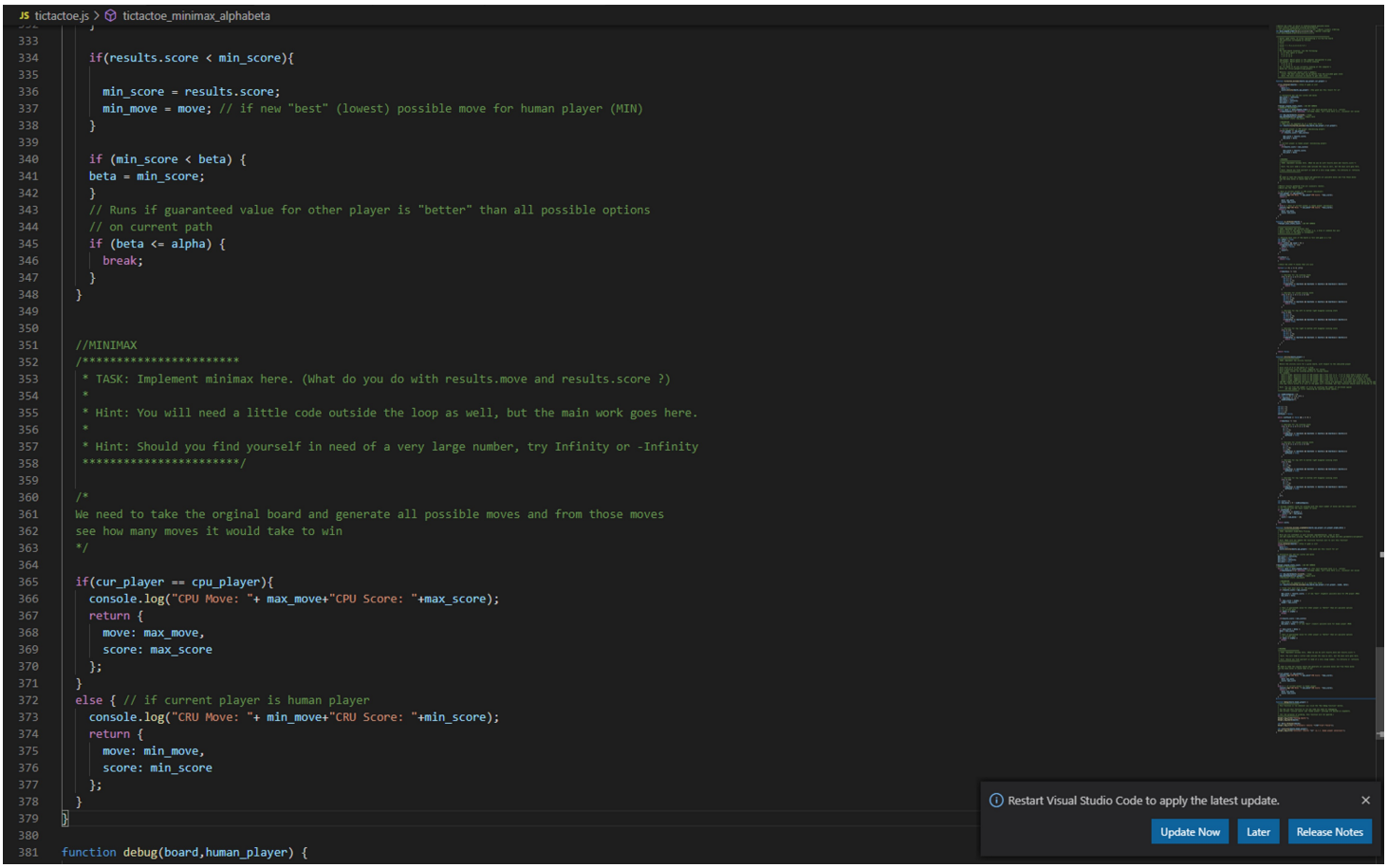
Restart Visual Studio Code to apply the latest update.

Update Now | Later | Release Notes

```
JS tictactoe.js > tictactoe_minimax_alphabeta
285       * TASK: Implement Alpha-Beta Pruning
286       *
287       * Once you are confident in your minimax implementation, copy it here
288       * and add alpha-beta pruning. (What do you do with the new alpha and beta parameters/variables?)
289       *
290       * Hint: Make sure you update the recursive function call to call this function!
291       **********************/
292       if(is_terminal(board)) //Stop if game is over
293       return {
294         move:null,
295         score:utility(board,cpu_player) //How good was this result for us?
296       }
297
298      // Initialize max and min scores and moves
299       max_score = -Infinity;
300       max_move = null;
301       min_score = Infinity;
302       min_move = null;
303
304     ++helper_expand_state_count; //DO NOT REMOVE
305     //GENERATE SUCCESSORS//
306       for(let move of move_expand_order) { //For each possible move (i.e., action)
307         if(board[move]!=-1) continue; //Already taken, can't move here (i.e., successor not valid)
308
309         let new_board=board.slice(0); //Copy
310         new_board[move]=cur_player; //Apply move
311         //Successor state: new_board
312
313         //RECURSION
314         // What will my opponent do if I make this move?
315         let results=tictactoe_minimax(new_board,cpu_player,1-cur_player, alpha, beta);
316
317         // Finds next best move for CPU player
318         if (results.score > max_score){
319
320           max_score = results.score; // if new "best" (highest) possible move for CPU player (MAX)
321           max_move = move;
322
323         }
324         if (max_score > alpha) {
325           alpha = max_score;
326         }
327
328         // Runs if guaranteed value for other player is "better" than all possible options
329         // on current path
330         if (beta <= alpha) {
331           break;
332         }
333
334         if(results.score < min_score){
```

Restart Visual Studio Code to apply the latest update.

Update Now | Later | Release Notes


tictactoe.js > tictactoe_minimax_alphabeta
if(results.score < min_score){
min_score = results.score;
min_move = move; // if new "best" (lowest) possible move for human player (MIN)
if (min_score < beta) {
beta = min_score;
// Runs if guaranteed value for other player is "better" than all possible options
// on current path
if (beta <= alpha) {
break;
//MINIMAX
* TASK: Implement minimax here. (What do you do with results.move and results.score ?)
* Hint: You will need a little code outside the loop as well, but the main work goes here.
* Hint: Should you find yourself in need of a very large number, try Infinity or -Infinity
We need to take the orginal board and generate all possible moves and from those moves
see how many moves it would take to win
if(cur_player == cpu_player){
console.log("CPU Move: "+ max_move+"CPU Score: "+max_score);
return {
move: max_move,
score: max_score
};
else { // if current player is human player
console.log("CRU Move: "+ min_move+"CRU Score: "+min_score);
return {
move: min_move,
score: min_score
};
function debug(board,human_player) {
Restart Visual Studio Code to apply the latest update.
Update Now
Later
Release Notes