

Hw-4

Sunday, September 27, 2020 9:30 PM

CSE3521_HW4

Authors: Lauren Saggard 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 not 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 as I 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 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: 294778 states

How do the two algorithms compare?

Results with new move_expand_order:

Nodes expanded with Minimax: Evaluated 549946 states, Expanded 294778 states

Nodes expanded with Minimax + Alpha-Beta Pruning: Evaluated 55506 states, Expanded 29634 states

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

The number of evaluate and expanded states dramatically decreased. This is because with Alpha Beta we are not looking at states that won't make a difference so we get to skip over them and reduce our total counts. So this works according to what we thought.

```
js tictactoe.js > ⓘ tictactoe_minimax_alphaBeta
1 //Define the order in which to examine/expand possible moves
2 // (This affects alpha-beta pruning performance)
3 // let move_expand_order=[0,1,2,3,4,5,6,7,8]; //Naive (linear) ordering
4 let move_expand_order=[4,0,1,2,3,5,6,7,8]; //Better ordering?
5 //let move_expand_order=[4,0,2,6,8,1,3,5,7]
6
7 ****
8 * board: game state, an array representing a tic-tac-toe board
9 * The positions correspond as follows
10 * 0|1|2
11 * -+-+
12 * 3|4|5 -> [ 0,1,2,3,4,5,6,7,8 ]
13 * -+-+
14 * 6|7|8
15 * For each board location, use the following:
16 * -1 if this space is blank
17 * 0 if it is X
18 * 1 if it is O
19 *
20 * cpu_player: Which piece is the computer designated to play
21 * cur_player: Which piece is currently playing
22 * 0 if it is X
23 * 1 if it is O
24 * So, to check if we are currently looking at the computer's
25 * moves do: if(cur_player==cpu_player)
26 *
27 * Returns: Javascript object with 2 members:
28 * score: The best score that can be gotten from the provided game state
29 * move: The move (location on board) to get that score
30 ****
31
32 function tictactoe_minimax(board,cpu_player,cur_player) {
33
34     if(is_terminal(board)) //Stop if game is over
35         return {
36             move:null,
37             score:utility(board,cpu_player) //How good was this result for us?
38         }
39
40     // Initialize max and min scores and moves
41     max_score = -Infinity;
42     max_move = null;
43     min_score = Infinity;
44     min_move = null;
45
46     ++helper_expand_state_count; //DO NOT REMOVE
47     //GENERATE SUCCESSORS//
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 }
```

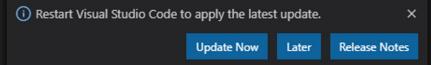
ⓘ Restart Visual Studio Code to apply the latest update.

Update Now Later Release Notes

```

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 original 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 successors (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 {

```

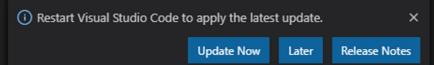


```

JS tictactoe.js > ⚡ tictactoe_minimax_alphaBeta
96     move: max_move,
97     score: max_score
98   };
99 } else { // Runs if current player is human player (minimizer)
100   console.log("CPU Move: "+ min_move+"CPU Score: "+min_score);
101   return {
102     move: min_move,
103     score: min_score
104   };
105 }
106 }
107 }
108 }
109 }

110 function isTerminal(board) {
111   ++helper_eval_state_count; //DO NOT REMOVE
112
113   /*****
114   * TASK: Implement the terminal test
115   * Return true if the game is finished (i.e., a draw or someone has won)
116   * Return false if the game is incomplete
117   *****/
118
119   // Checking base case if the board is full and game is a tie
120   let isFull = true;
121   let count = 0;
122   while (isFull && count < 9) {
123     if(board[count] == -1){
124       isFull = false;
125     } else {
126       count++;
127     }
128   }
129
130   if(isFull) {
131     return true;
132   }
133
134   //Check the other 9 states that are wins
135
136   for(var j = 0; j <= 6; j++){
137
138     if(board[j] != -1{
139
140       // Searches for row winning state
141       if(j == 0 || j == 3 || j == 6){
142         let a = j;
143         let b = j +1;
144

```



```
JS ticactoe.js > ticactoe_minimax_alphaBeta
143     let a = j;
144     let b = j +1;
145     let c = j+2;
146     if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
147         return true;
148     }
149 }
150
151 // Searches for column winning state
152 if(j == 0 || j == 1 || j == 2){
153     let a = j;
154     let b = j +3;
155     let c = j+6;
156     if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
157         return true;
158     }
159 }
160
161 // Searches for top left to bottom right diagonal winning state
162 if(j == 0){
163     let a = j;
164     let b = j +4;
165     let c = j+8;
166     if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
167         return true;
168     }
169 }
170
171 // Searches for top right to bottom left diagonal winning state
172 if(j == 2){
173     let a = j;
174     let b = j +2;
175     let c = j+4;
176     if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
177         return true;
178     }
179 }
180
181 }
182
183
184 return false;
185 }
186
187 function utility(board,player) {
188     ****
189     * TASK: Implement the utility function
190     *
191     * Return the utility score for a given board, with respect to the indicated player
192     *
193 }
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
```

Restart Visual Studio Code to apply the latest update.

Update Now Later Release Notes

```
JS ticactoe.js > ticactoe_minimax_alphaBeta
190
191     * Return the utility score for a given board, with respect to the indicated player
192     *
193     * Give score of 0 if the board is a draw
194     * Give a positive score for wins, negative for losses.
195     * Give larger scores for winning quickly or losing slowly
196     * For example:
197     *   Give a large, positive score if the player had a fast win (i.e., 5 if it only took 5 moves to win)
198     *   Give a small, positive score if the player had a slow win (i.e., 1 if it took all 9 moves to win)
199     *   Give a small, negative score if the player had a slow loss (i.e., -1 if it took all 9 moves to lose)
200     *   Give a large, negative score if the player had a fast loss (i.e., -5 if it only took 5 moves to lose)
201     * (DO NOT simply hard code the above 4 values, other scores are possible. Calculate the score based on the above pattern.)
202     * (You may return either 0 or null if the game isn't finished, but this function should never be called in that case anyways.)
203     *
204     * Hint: You can find the number of turns by counting the number of non-blank spaces
205     *       (Or the number of turns remaining by counting blank spaces..)
206     ****
207
208     var numBlankSpaces = 0;
209     for (var i = 0; i < 9; i++) {
210         if (board[i] == -1) {
211             numBlankSpaces++;
212         }
213     }
214
215     let a = -1;
216     let b = -1;
217     let c = -1;
218     let j = 0;
219     setFound = false;
220
221     while (setFound == false && j <= 6) {
222
223         if(board[j] != -1){
224
225             // Searches for row winning state
226             if(j == 0 || j == 3 || j == 6){
227                 a = j;
228                 b = j +1;
229                 c = j+2;
230                 if(board[a] == board[b] && board[b] == board[c] && board[a]== board[c]){
231                     setFound = true;
232                 }
233             }
234
235             // Searches for column winning state
236             if(j == 0 || j == 1 || j == 2){
237                 a = j;
238                 b = j +3;
239                 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     *

```

➊ 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         if(results.score < min_score){


```

➊ Restart Visual Studio Code to apply the latest update. Update Now Later Release Notes

```
JS tic tac toe.js > ⚡ tic tac toe_minimax_alpha beta
333
334     if(results.score < min_score){
335
336         min_score = results.score;
337         min_move = move; // if new "best" (lowest) possible move for human player (MIN)
338     }
339
340     if (min_score < beta) {
341         beta = min_score;
342     }
343     // Runs if guaranteed value for other player is "better" than all possible options
344     // on current path
345     if (beta <= alpha) {
346         break;
347     }
348 }
349
350
351 //MINIMAX
352 /***** */
353 * TASK: Implement minimax here. (What do you do with results.move and results.score ?)
354 *
355 * Hint: You will need a little code outside the loop as well, but the main work goes here.
356 *
357 * Hint: Should you find yourself in need of a very large number, try Infinity or -Infinity
358 *****/
359
360 /*
361 We need to take the original board and generate all possible moves and from those moves
362 see how many moves it would take to win
363 */
364
365 if(cur_player == cpu_player){
366     console.log("CPU Move: "+ max_move+"CPU Score: "+max_score);
367     return {
368         move: max_move,
369         score: max_score
370     };
371 }
372 else { // if current player is human player
373     console.log("HUMAN Move: "+ min_move+"HUMAN Score: "+min_score);
374     return {
375         move: min_move,
376         score: min_score
377     };
378 }
379
380
381 function debug(board,human_player) {
```

Restart Visual Studio Code to apply the latest update.

[Update Now](#) [Later](#) [Release Notes](#)