Jeremy Grifski

09/26/19

CSE 5521

Homework 4

1. Game playing.
   1. Implement terminal check and utility functions for tic-tac-toe. See the is\_terminal and utility functions in tictactoe.js for more detailed instructions. (2 pts)

**Done.**

* 1. Create a function to play tic-tac-toe, using the Minimax algorithm. Modify the template function tictactoe\_minimax, provided in tictactoe.js, to accomplish this. (Note: You have already been provided a working recursive depth-first search implementation; you need only add Minimax to it.) (3 pts)

**Done.**

* 1. Use the provided tictactoe.htm file to play a few games against your code from (1b). Does it play the way you expect? Can you beat it? Also, notice that the evaluated/expanded state counts are reported while you are playing a game. Record these values as you play. How do they change over the course of the game? Is this what you expected? Discuss. (1 pt)

**Since the computer assumes that I’ll take the best action, the computer plays optimally. As a result, I can’t beat it. We always tie (or I lose).**

**If I take the first turn, the following states are expanded and evaluated, respectively:**

**59705 – 31973**

**1053 – 533**

**47 – 26**

**5 – 3**

**Naturally, the number of expanded states reduces as the game continues because there is less of the world to explore and anticipate.**

* 1. Modify your algorithm to use alpha-beta pruning. (Copy your code from (1b) into the provided tictactoe\_minimax\_alphabeta function. Modify this.) (3 pts)

**Done.**

* 1. Repeat (1c) using your new function. (1pt)

**Again, the game is unbeatable. Of course, now, the computer is much faster at making a decision:**

**4077 – 2368**

**436 – 231**

**43 – 24**

**5 – 3**

**Surprisingly, the alpha beta algorithm seems to have diminishing returns. In other words, the effectiveness is reduced as the number of possible states is reduced.**

* 1. Configure the game so the human player goes second (and make sure the initial board is empty). Set for standard MiniMax and start the game. Report how many nodes are expanded to calculate the first move. Repeat for MiniMax w/ Alpha-Beta Pruning. How do the two algorithms compare? Are the results what you expected? Discuss. (continued) Now, look again in tictactoe.js and notice the variable move\_expand\_order, which controls the order in which different possible moves are examined. An alternate ordering has been provided but commented out. Uncomment this and repeat the above experiment. How do the results change? Try to explain why you get this result. (1pt)
* **MiniMax Evaluated States: 549946**
* **MiniMax Expanded States: 294778**
* **AlphaBeta Evaluated States: 23489**
* **AlphaBeta Expanded States: 13781**

**Naturally, AlphaBeta performs exponentially better than MiniMax. Since AlphaBeta tends to cut tree height in half, I’m not really surprised, but I do think it’s interesting.**

**With the expand order changed, we see the following stats:**

* **MiniMax Evaluated States: 549946**
* **MiniMax Expanded States: 294778**
* **AlphaBeta Evaluated States: 13595**
* **AlphaBeta Expanded States: 7764**

**With these results, we see a dramatic improvement for the AlphaBeta solution. In this example, the algorithm starts with the middle element as opposed to the top left element. Apparently, this ordering is better, but I’m a bit stumped as to why.**

**My theory is that placing an O at the center location dramatically reduces the number of win conditions. Instead of 8, there are now 4. With a corner placement, the win states are only reduced by 2. As a result, the optimal ordering would be to check states that limit the most win conditions. For instance, something like 4, 0, 2, 6, 8, 1, 3, 5, 7.**

**EXTRA CREDIT: based on my theory above, I was able to confirm that my ordering was better. See the following results:**

* **AlphaBeta Evaluated States: 9003**
* **AlphaBeta Expanded States: 5318**

**I suspect that you may be able to reduce the expanded states even further by checking diagonals in a different order. Of course, this was good enough for me. That said, I tried flipping the order by doing sides then corners then center: 1, 3, 5, 7, 0, 2, 6, 8, 4. Perhaps that way, I could get an even worse result:**

* **AlphaBeta Evaluated States: 52860**
* **AlphaBeta Expanded States: 30756**

**Clearly, this ordering is the worst so far. Unfortunately, I wasn’t able to explore any further to confirm that it was indeed the worst solution.**

1. Use truth tables to show whether or not the following sentences are valid (make sure to state whether or not it is valid!). (2 pts each)

**INVALID! Not all statements are true.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P | Q | P ۸ Q | ¬Q⇒P | [ P ۸ Q ] ⇔ [ ¬Q⇒P ] |
| T | T | T | T | T |
| T | F | F | F | T |
| F | T | F | T | F |
| F | F | F | T | F |

**VALID! All statements are true.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| P | Q | R | P ⇒ (Q ۷ R) | ¬P ۷ Q ۷ R | [ P ⇒ (Q ۷ R) ] ⇔ [ ¬P ۷ Q ۷ R ] |
| T | T | T | T | T | T |
| T | T | F | T | T | T |
| T | F | T | T | T | T |
| T | F | F | F | F | T |
| F | T | T | T | T | T |
| F | T | F | T | T | T |
| F | F | T | T | T | T |
| F | F | F | T | T | T |

Appendix

//Define the order in which to examine/expand possible moves

//(This affects alpha-beta pruning performance)

//let move\_expand\_order = [0, 1, 2, 3, 4, 5, 6, 7, 8]; //Naive (linear) ordering

//let move\_expand\_order = [4, 0, 1, 2, 3, 5, 6, 7, 8]; //Better ordering?

let move\_expand\_order = [4, 0, 2, 6, 8, 1, 3, 5, 7]; // Optimal ordering

//let move\_expand\_order = [1, 3, 5, 7, 0, 2, 6, 8, 4]; // Worst ordering

// A mapping of all possible win conditions

const mapping = [

  [0, 1, 2],

  [3, 4, 5],

  [6, 7, 8],

  [0, 3, 6],

  [1, 4, 7],

  [2, 5, 8],

  [0, 4, 8],

  [2, 4, 6]

]

/////////////////////////////////////////////////////////////////////////////

function tictactoe\_minimax(board, cpu\_player, cur\_player) {

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  \* board: game state, an array representing a tic-tac-toe board

  \* The positions correspond as follows

  \* 0|1|2

  \* -+-+-

  \* 3|4|5 -> [ 0,1,2,3,4,5,6,7,8 ]

  \* -+-+-

  \* 6|7|8

  \* For each board location, use the following:

  \*  -1 if this space is blank

  \*   0 if it is X

  \*   1 if it is O

  \*

  \* cpu\_player: Which piece is the computer designated to play

  \* cur\_player: Which piece is currently playing

  \*   0 if it is X

  \*   1 if it is O

  \* So, to check if we are currently looking at the computer's

  \* moves do: if(cur\_player===cpu\_player)

  \*

  \* Returns: Javascript object with 2 members:

  \*   score: The best score that can be gotten from the provided game state

  \*   move: The move (location on board) to get that score

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  //BASE CASE

  if (is\_terminal(board)) //Stop if game is over

    return {

      move: null,

      score: utility(board, cpu\_player) //How good was this result for us?

    }

  let min\_max\_score = cur\_player === cpu\_player ? -Infinity : Infinity;

  let min\_max\_move = null;

  ++helper\_expand\_state\_count; //DO NOT REMOVE

  //GENERATE SUCCESSORS

  for (let move of move\_expand\_order) { //For each possible move (i.e., action)

    if (board[move] != -1) continue; //Already taken, can't move here (i.e., successor not valid)

    let new\_board = board.slice(0); //Copy

    new\_board[move] = cur\_player; //Apply move

    //Successor state: new\_board

    //RECURSION

    // What will my opponent do if I make this move?

    let results = tictactoe\_minimax(new\_board, cpu\_player, 1 - cur\_player);

    //MINIMAX

    if (cur\_player === cpu\_player) {

      if (results.score > min\_max\_score) {

        min\_max\_score = results.score;

        min\_max\_move = move;

      }

    } else {

      if (results.score < min\_max\_score) {

        min\_max\_score = results.score;

        min\_max\_move = move;

      }

    }

  }

  //Return results gathered from all sucessors (moves).

  //Which was the "best" move?

  return {

    move: min\_max\_move,

    score: min\_max\_score

  };

}

/\*\*

 \* Tests if the game is complete.

 \*

 \* @param {Array} board that game board

 \*/

function is\_terminal(board) {

  ++helper\_eval\_state\_count; //DO NOT REMOVE

  let winner = get\_win\_value(board);

  let isFull = is\_full(board);

  return winner >= 0 || isFull;

}

/\*\*

 \* A helper function which detects if the board

 \* is filled or not.

 \*

 \* @param {Array} board the game board

 \*/

function is\_full(board) {

  let count = 0;

  for (var i = 0; i < board.length; i++) {

    if (board[i] != -1) {

      count++;

    }

  }

  return count == board.length;

}

/\*\*

 \* Returns which player won:

 \* 0 for X

 \* 1 for O

 \* -1 if neither (not necessarily a draw)

 \*

 \* @param {Array} board the game board as a list

 \*/

function get\_win\_value(board) {

  for (var i = 0; i < mapping.length; i++) {

    var check = [];

    for (var j = 0; j < mapping[i].length; j++) {

      check.push(board[mapping[i][j]]);

    }

    if (check.every((val, i, arr) => val === arr[0] && val != -1)) {

      return check[0];

    }

  }

  return -1;

}

/\*\*

 \* TASK: Implement the utility function

 \*

 \* 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.)

 \*

 \* @param {Array} board the game board

 \* @param {Number} player the current player

 \*/

function utility(board, player) {

  // Count the number of turns

  var turnsLeft = 0;

  for (var i = 0; i < board.length; i++) {

    if (board[i] == -1) {

      turnsLeft += 1;

    }

  }

  // Determine if we have a winner

  var score = 0;

  let winner = get\_win\_value(board);

  if (winner >= 0) {

    score = turnsLeft + 1;

    if (winner != player) {

      score \*= -1;

    }

  }

  return score;

}

function tictactoe\_minimax\_alphabeta(board, cpu\_player, cur\_player, alpha, beta) {

  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  \* TASK: Implement Alpha-Beta Pruning

  \*

  \* Once you are confident in your minimax implementation, copy it here

  \* and add alpha-beta pruning. (What do you do with the new alpha and beta parameters/variables?)

  \*

  \* Hint: Make sure you update the recursive function call to call this function!

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  //BASE CASE

  if (is\_terminal(board)) //Stop if game is over

    return {

      move: null,

      score: utility(board, cpu\_player) //How good was this result for us?

    }

  let min\_max\_score = cur\_player === cpu\_player ? -Infinity : Infinity;

  let min\_max\_move = null;

  ++helper\_expand\_state\_count; //DO NOT REMOVE

  //GENERATE SUCCESSORS

  for (let move of move\_expand\_order) { //For each possible move (i.e., action)

    if (board[move] != -1) continue; //Already taken, can't move here (i.e., successor not valid)

    let new\_board = board.slice(0); //Copy

    new\_board[move] = cur\_player; //Apply move

    //Successor state: new\_board

    //RECURSION

    // What will my opponent do if I make this move?

    let results = tictactoe\_minimax\_alphabeta(new\_board, cpu\_player, 1 - cur\_player, alpha, beta);

    //MINIMAX

    if (cur\_player === cpu\_player) {

      if (results.score > alpha) {

        min\_max\_score = results.score;

        alpha = results.score;

        min\_max\_move = move;

      }

    } else {

      if (results.score < beta) {

        min\_max\_score = results.score;

        beta = results.score;

        min\_max\_move = move;

      }

    }

    if (alpha > beta) {

      break;

    }

  }

  //Return results gathered from all sucessors (moves).

  //Which was the "best" move?

  return {

    move: min\_max\_move,

    score: min\_max\_score

  };

}