```
//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]; //Even better??
// let move_expand_order=[7,5,3,1,8,6,2,0,4]; //Worse??

/* AUTHORS: Austin Schall.37 and John Choi.1655 */

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

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?
    }

  ++helper_expand_state_count; //DO NOT REMOVE
  //GENERATE SUCCESSORS

  var best_score = Infinity
  if(cur_player == cpu_player){
    best_score = -Infinity;
  }
  var nextMove = -1
  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
    /***********************
    * 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
    ***********************/
    if ((cur_player == cpu_player) && (results.score > best_score)) {
      best_score = results.score;
      nextMove = move;
    }
    if ((cur_player != cpu_player) && (results.score < best_score)){
      best_score = results.score;
      nextMove = move;
    }
  }

  //Return results gathered from all sucessors (moves).
  //Which was the "best" move?
  return {
    move: nextMove/* What do you return here? */,
    score: best_score/* And here? */
  };
}

function win_exists(board) {
  // check vertical
  for (var i = 0; i < 3; i++) {
    if (board[i] == -1) {
      continue
    }
    if (board[i] == board[i + 3] && board[i] == board[i + 6]) {
      return true
    }
  }
  // check horizontal
  for (var i = 0; i < 3; i++) {
    let numVal = board[3 * i]
    var all_equals = true
    for (var j = 3 * i; j <= 3 * i + 2; j++) {
      // if all 3 numbers across the board horizontally are equal to each
       other, return true
      if (numVal == -1 || board[j] != numVal) {
        all_equals = false
      }
    }
    // if all_equals is still true at this point, all 3 were equal
    if (all_equals) {
      return true
    }
  }
  // check diagonal
  return (board[0] != -1 && board[0] == board[4] && board[0] == board[8]) ||
            (board[2] != -1 && board[2] == board[4] && board[2] == board[6])
}

function is_terminal(board) {
  ++helper_eval_state_count; //DO NOT REMOVE

  /*************************
  * TASK: Implement the terminal test
  * Return true if the game is finished (i.e, a draw or someone has won)
  * Return false if the game is incomplete
  *************************/
  // check for win
  if (win_exists(board)) {
    return true
  }
  // check if incomplete
  // if there's any -1 in the board, it is incomplete
  for (var i = 0; i < 9; i++) {
    if (board[i] == -1) {
      return false
    }
  }
  // by the time we get here, we know that there is a draw
  return true
}

function utility(board,player) {
  /***********************
  * 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.)
***********************/

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


let winner = -1;
/* Win Exists */
for (var i = 0; i < 3; i++) {
  if (board[i] == -1) {
    continue
  }
  if (board[i] == board[i + 3] && board[i] == board[i + 6]) {
    winner = board[i];
  }
}

// Horizontal check
for(var i = 0; i < 9; i+=3){
  if (board[i] == -1) {
    continue
  }
  if (board[i] == board[i + 1] && board[i] == board[i + 2]) {
    winner = board[i];
    }
  }

  // check diagonal
  if((board[0] != -1 && board[0] == board[4] && board[0] == board[8]) ||
   (board[2] != -1 && board[2] == board[4] && board[2] == board[6])){
    winner = board[4];
  }

  /* If there is a draw */
  if(winner == -1){
    return 0;
  }

  var score = 1 + blanks;
  if(player == winner){
    return score;
  } else{
    return (score * -1);
  }


}

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?
    }

  ++helper_expand_state_count; //DO NOT REMOVE
  //GENERATE SUCCESSORS

  var best_score = Infinity
  if(cur_player == cpu_player){
    best_score = -Infinity;
  }
```

```
var nextMove = -1
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
  /***********************
  * 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
  ***********************/
  if ((cur_player == cpu_player) && (results.score > best_score)) {
    best_score = results.score;
    nextMove = move;
  }
  if ((cur_player != cpu_player) && (results.score < best_score)){
    best_score = results.score;
    nextMove = move;
  }

  if(cur_player == cpu_player){
    alpha = Math.max(alpha, best_score);
  } else{
    beta = Math.min(beta, best_score);
  }

  if (alpha > beta){
    break;
  }
}

//Return results gathered from all sucessors (moves).
//Which was the "best" move?
return {
  move: nextMove/* What do you return here? */,
  score: best_score/* And here? */
};
```

```
}

function debug(board,human_player) {
  /***********************
  * This function is run whenever you click the "Run debug function" button.
  *
  * You may use this function to run any code you need for debugging.
  * The current "initial board" and "human player" settings are passed as
   arguments.
  *
  * (For the purposes of grading, this function will be ignored.)
  ***********************/
  helper_log_write("Testing board:");
  helper_log_board(board);

  let tm=is_terminal(board);
  helper_log_write("is_terminal() returns "+(tm?"true":"false"));

  let u=utility(board,human_player);
  helper_log_write("utility() returns "+u+" (w.r.t. human player selection)");
}
```