```
/*
================
8-puzzle problem
================
Construct a 3x3 grid, containing one blank(empty) space and one each of tiles
 labeled 1-8.
By exchanging tiles adjacent to the blank space with the blank space, achieve
 the desired configuration:
 1 2 3
 8   4
 7 6 5

State:
{
  grid : Array(3,3), Integers [0,8]
}
where grid is a 2D array whose indices correspond to the following grid
 positions:
 [0][0] [0][1] [0][2]
 [1][0] [1][1] [1][2]
 [2][0] [2][1] [2][2]
The value 0 is used to represent the blank space, and 1-8 for the
 corresponding labeled tiles.

Possible actions:
ID | Action
---+----------------------
 1 | Move tile above blank down (i.e., "move" blank up)
---+----------------------
 2 | Move tile below blank up (i.e., "move" blank down)
---+----------------------
 3 | Move tile left of blank right (i.e., "move" blank left)
---+----------------------
 4 | Move tile right of blank left (i.e., "move" blank right)
*/

///////////////////////////////////////////////////////////////////////////////
// Complete the following two functions

/**
 * AUTHORS: John Choi and Austin Schall
 */

//Check if the given state is a goal state
//Returns: true if is goal state, false otherwise
function is_goal_state(state) {
  ++helper_eval_state_count; //Keep track of how many states are evaluated (DO
   NOT REMOVE!)

  return state.grid[0][0] == 1 && state.grid[0][1] == 2 && state.grid[0][2] ==
   3 &&
     state.grid[1][2] == 4 && state.grid[2][2] == 5 && state.grid[2][1] == 6 &&
     state.grid[2][0] == 7 && state.grid[1][0] == 8 && state.grid[1][1] == 0;
}

//Find the list of actions that can be performed from the given state and the
 new
//states that result from each of those actions
//Returns: Array of successor objects (where each object has a valid actionID
 member and corresponding resultState member)
function find_successors(state) {
  ++helper_expand_state_count; //Keep track of how many states are expanded
   (DO NOT REMOVE!)

  let successors = [];

  /***Your code to generate successors here!***/
  // find location of blank
  var i_idx = 0, j_idx = 0;
  var terminateLoop = false; // variable to terminate outer for loop when
   blank is found
  for (var i = 0; i < 3 && !terminateLoop; i++) {
    for (var j = 0; j < 3; j++) {
      if (state.grid[i][j] == 0) {
        i_idx = i;
        j_idx = j;
        terminateLoop = true;
        break;
      }
    }
  }

  // check if blank can go up
  if (i_idx - 1 >= 0) {
    // can go up

    var newGrid = state.grid.map(x => x.slice(0));
    let aboveVal = newGrid[i_idx - 1][j_idx];
    newGrid[i_idx - 1][j_idx] = 0;
    newGrid[i_idx][j_idx] = aboveVal;

    // make new state
    let newState = { // TODO: what does x => x.slice(0) mean? how do we update
     the grid state?
      grid: newGrid
    };

    // append to successors list
    successors.push( {
      actionID: 1,
      resultState: newState
    });
```

```
}
// check if blank can go left
if (j_idx - 1 >= 0) {
  // can go left

  // copy grid
  var newGrid = state.grid.map(x => x.slice(0));
  let leftVal = newGrid[i_idx][j_idx - 1];
  newGrid[i_idx][j_idx - 1] = 0;
  newGrid[i_idx][j_idx] = leftVal;

  // make new state
  let newState = {
    grid: newGrid
  };

  // append to successors list
  successors.push( {
    actionID: 3,
    resultState: newState
  });
}
// check if blank can go right
if (j_idx + 1 < 3) {
  // can go right

  // copy grid
  var newGrid = state.grid.map(x => x.slice(0));
  let rightVal = newGrid[i_idx][j_idx + 1];
  newGrid[i_idx][j_idx + 1] = 0;
  newGrid[i_idx][j_idx] = rightVal;

  // make new state
  let newState = {
    grid: newGrid
  };

  // append to successors list
  successors.push( {
    actionID: 4,
    resultState: newState
  });
}
// check if blank can go down
if (i_idx + 1 < 3) {
  // can go down

  // copy grid
  var newGrid = state.grid.map(x => x.slice(0));
  let belowVal = newGrid[i_idx + 1][j_idx];
  newGrid[i_idx + 1][j_idx] = 0;
```

```
    newGrid[i_idx][j_idx] = belowVal;

    // make new state
    let newState = {
      grid : newGrid
    };

    // append to successors list
    successors.push( {
      actionID: 2,
      resultState: newState
    });
  }

  //Hint: Javascript objects are passed by reference, so don't modify "state"
   directy.
  //Make copies instead:
  //  let newState={
  //    grid : state.grid.map(x => x.slice(0)) //Deep copy of grid
  //  };
  //Remember to make a new copy for each new state you make!

  //Hint: Add new elements to the successor list like so:
  //  successors.push({
  //    actionID : /*ID*/,
  //    resultState : newState
  //  });

  return successors;
}

////////////////////////////////////////////////////////////////////////////////
// Use these functions when developing your A* implementation

//Heuristic functions for the 8-puzzle problem
function calculate_heuristic(state) {
  //Total Manhattan distance heuristic
  let goal=[ [1, 2, 3], [8, 0, 4], [7, 6, 5] ];

  let g_pos=Array(9);
  let st_pos=Array(9);
  for(let j=0;j<3;++j)
    for(let i=0;i<3;++i) {
        g_pos[ goal[j][i] ]=[j,i];
        st_pos[ state.grid[j][i] ]=[j,i];
    }

  let h=0;
  for(let i=0;i<9;++i) {
    h+=Math.abs( st_pos[i][0]-g_pos[i][0] )+Math.abs( st_pos[i][1]-g_pos[i][1]
     );
  }
  return h;
}

/*
function calculate_heuristic(state) {
  //Misplaced tiles heuristic
  let goal=[ [1, 2, 3], [8, 0, 4], [7, 6, 5] ];

  let h=0;
  for(let j=0;j<3;++j)
    for(let i=0;i<3;++i) {
      if(state.grid[j][i]!=goal[j][i])
        ++h;
    }
  if(h>0) --h; //Account for miscounted blank
  return h;
}
*/

/*
function calculate_heuristic(state) {
  //Simplest heuristic (h(n)=0)
  return 0;
}
*/
```