```
//Perform breadth-first search from initial state, using defined
 "is_goal_state"
//and "find_successors" functions
//Returns: null if no goal state found
//Returns: object with two members, "actions" and "states", where:
//  actions: Sequence(Array) of action ids required to reach the goal state
 from the initial state
//  states: Sequence(Array) of states that are moved through, ending with the
 reached goal state (and EXCLUDING the initial state)
//  The actions and states arrays should both have the same length.

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

function astar_search(initial_state) {
  let open = new FastPriorityQueue(function(a,b) { return
   a.estimated_total_cost < b.estimated_total_cost; });
  let closed = new Set();
  let fixed_step_cost = 1; //Assume action cost is constant

  /***Your code for A* search here***/

  var currentNode = {
    currentState: initial_state,
    children: find_successors(initial_state)
  }

  var currentNodeHistory = []

  var totalCost = 0
  while (!is_goal_state(currentNode.currentState)) {
    // add current node to the closed set
    closed.add(state_to_uniqueid(currentNode.currentState))
    // reset Priority Queue
    while (!open.isEmpty()) {
      open.poll()
    }
    // evaluate children
    let children = currentNode.children
    // iterate through children
    for (var i = 0; i < children.length; i++) {
      // make sure the child was not visited earlier
      if (!closed.has(state_to_uniqueid(children[i].resultState))) {
        open.add({
          estimated_total_cost: totalCost +
           calculate_heuristic(children[i].resultState),
          index: i
        })
      }
    }
```

```
    // Get the smallest child
    let cheapestChild = open.poll()
    // add next node to the history
    currentNodeHistory.push({
      action: currentNode.children[cheapestChild.index].actionID,
      state: currentNode.children[cheapestChild.index].resultState
    })
    // Update current node to be the cheapest child
    let currentNodeCopy = Object.assign({}, currentNode)
    let nextState = currentNodeCopy.children[cheapestChild.index].resultState
    let nextChildren =
     find_successors(currentNodeCopy.children[cheapestChild.index].resultState)
    currentNode = {
      currentState: nextState,
      children: nextChildren
    }
    // going to the next level in tree so increase the total cost
    totalCost += fixed_step_cost
  }

  /*
    Hint: A* is very similar to BFS, you should only need to make a few small
     modifications to your BFS code.

    You will need to add values to your augmented state for path cost and
     estimated total cost.
    I suggest you use the member name "estimated_total_cost" so that the above
     priority queue code will work.

    Call function calculate_heuristic(state) (provided for you) to calculate
     the heuristic value for you.

    See (included) FastPriorityQueue.js for priority queue usage example.
  */
  var actionsToGoal = []
  var statesToGoal = []
  for (var i = 0; i < currentNodeHistory.length; i++) {
    let node = currentNodeHistory[i]
    actionsToGoal.push(node.action)
    statesToGoal.push(node.state)
  }
  return {
    actions : actionsToGoal,
    states : statesToGoal
  }
}
```