```
//Perform depth-limited search from initial state, using defined
 "is_goal_state"
//and "find_successors" functions
//Will not examine paths longer than "depth_limit" (i.e. paths that have
 "depth_limit" states in them, or "depth_limit-1" actions in them)
//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 depth_limited_search(initial_state,depth_limit) {
  let current_state = {
    state : initial_state,
    predecessor : null,
    action : null
  }

  return dls(current_state, depth_limit);
}

function dls(current_state, depth){
  /* If the current state is in the goal state, return them */
  if(is_goal_state(current_state.state)){
    let states = [];
    let actions = [];

    while (current_state.predecessor != null){
      states.push(current_state.state);
      actions.push(current_state.action);
      current_state = current_state.predecessor;
    }

    actions.reverse();
    states.reverse();

    return {
      states : states,
      actions : actions
    }
  /* If the depth is 0, return that no goal state found */
  } else if(depth == 0){
    return null;
  /* Run the recursive call with finding the successors from the current
   state. */
  } else{
    let sucs = find_successors(current_state.state);

    for(let i = 0; i < sucs.length; i++){
      let temp = {
        state : sucs[i].resultState,
        predecessor : current_state,
        action : sucs[i].actionID
      }

      temp = dls(temp, depth - 1);

      if (temp != null){
        return temp;
      }
    }


    return null;
  }
}


/***DO NOT do repeated state or loop checking!***/

  /*
    Hint: You may implement DLS either iteratively (with open set) or
     recursively.

    In the iterative case, you will need to do similar to breadth-first search
     and augment
    the state. In addition to predecessor and action, you will also need to
     store depth.
    (You should be able to re-use your BFS code and only make a small amount
     of changes to
     accomplish this. Be sure to remove repeat checking!)

    In the recursive case, you don't need the above. Building the solution
     path is a little
    trickier, but I suggest you look into the Array.unshift() function.
  */
```