Artificial Intelligence (CS 534)
Assignment 2

1.

The graph separation property propagates the idea that every path from the initial node to the unexplored node must pass through one of the nodes in the frontier, i.e., the leaf node.

Graph search satisfies this property.

At the start of the search, when no node has been explored, the frontier acts as the initial state. Hence, every path from the initial state to the unexplored state includes the initial state, which in this case is the frontier.

Let's assume that the frontier is not empty and the recently explored node is not the goal state. At the end of the iteration, the recently explored node would be removed from the frontier and its successors (the ones which have not been explored or already in the frontier) would be added to the frontier.

The property will hold true for graph searches because as you complete iterations and go through nodes looking for the goal state, the frontier node is always going to exist before going onto an unexplored state or node. As you perform the graph search, by the definition of the graph separation property, each path from the initial state to the unexplored state has a node in the frontier.

This property is violated by algorithms that move nodes from frontier to the explored state before their successors have been generated. It is possible to create search algorithms that will only generate certain nodes in order to get from node a to b, i.e., without having to generate all successors of a node at once before expanding another node, and this would violate the graph separation property.

2.

a.

Initial state: one randomly selected piece from the given pieces
Successor function: for any open track piece, add any piece type from remaining types. For a curved piece, add in either orientation; for a fork, add in either orientation and (if there are two holes) connecting at either hole. It's a good idea to disallow any overlapping configuration.
Goal test: all pieces used in a single connected track, no open pegs or holes, no overlapping tracks.
Step cost: one per piece

b.

Since all the solutions are at the same depth, so depth-first search would be appropriate.

c. A solution has no open pegs or holes, so every peg is in a hole, so there must be equal numbers of pegs and holes. Removing a fork violates this property. Hence, removing a fork makes this problem unsolvable.

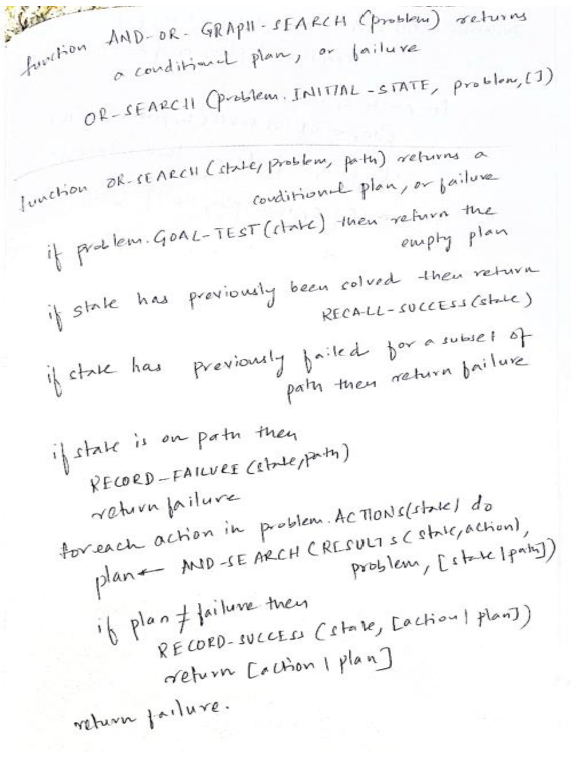
d. The maximum possible number of open pegs is 3 (starts at 1, adding a two-peg fork increases it by one). Assuming each piece is unique, any piece can be added to a peg, giving at most 12 + (2 * 16) + (2 * 2) + (2 * 2* 2) = 56 choices per peg.
The total depth is 32 since there are 32 pieces. So, the upper bound on the total size of the state space is 168^{32}/ (12! * 16! * 2! * 2!) where the factorials deal with the permutations.

3.

a. The branching factor is 4. From every node, there is a possibility of exploring 4 other nodes, given they were in the unexplored state.
b. There are a linear number of states along the boundary of the square. Hence, the answer would be 4k, where k is the depth.
c. BFS expands exponentially many nodes: counting precisely, we get $((4^{x+y+1} - 1)/3) - 1$.
d. There are quadratically many states within the square for depth x + y. Hence, the answer is $2(x + y)*(x + y + 1) - 1$.
e. The given h function is the Manhattan distance metric. Yes, it is an admissible heuristic.
f. All nodes in the rectangle defined by (0, 0) and (x, y) are candidates for the optimal path, and there are quadratically many of them.
g. Yes. By admissible, we are asking if h is an underestimate. Removing links would cause detours but it wont cause an over-estimate.
h. False; nonlocal links can reduce the actual path length below the Manhattan distance.

5.

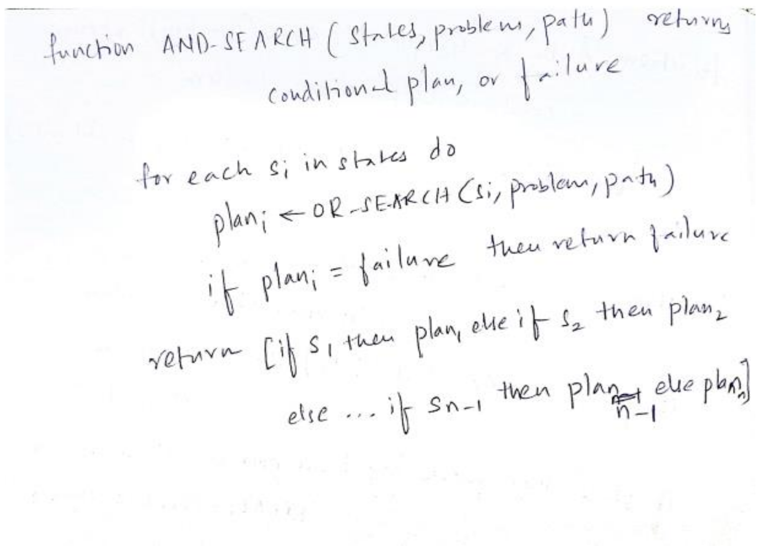
Updated algorithm:



```
function AND-OR-GRAPH-SEARCH (problem) returns
        a conditional plan, or failure
    OR-SEARCH (problem.INITIAL-STATE, problem, [])

function OR-SEARCH (state, problem, path) returns a
        conditional plan, or failure
    if problem.GOAL-TEST(state) then return the
        empty plan
    if state has previously been colved then return
        RECALL-SUCCESS(state)
    if state has previously failed for a subset of
        path then return failure

    if state is on path then
        RECORD-FAILURE (state, path)
        return failure
    foreach action in problem.ACTIONS(state) do
        plan ← AND-SEARCH (RESULTS(state, action),
                problem, [state | path])
        if plan ≠ failure then
            RECORD-SUCCESS (state, [action | plan])
            return [action | plan]
    return failure.
```



```
function AND-SEARCH (states, problem, path) returns
        conditional plan, or failure

    for each $s_i$ in states do
        $plan_i$ ← OR-SEARCH($s_i$, problem, path)
        if $plan_i$ = failure then return failure
    return [if $s_1$ then $plan_1$ else if $s_2$ then $plan_2$
        else ... if $s_{n-1}$ then $plan_{n-1}$ else $plan_n$]
```

Reasoning:

For states that OR-SEARCH finds a solution for it records the solution found. If it later visits that state again it immediately returns that solution. When OR-SEARCH fails to find a solution, it has to be careful. Whether a state can be solved depends on the path taken to that solution, as we do not allow cycles.

So, on failure OR-SEARCH records the value of path. If a state is which has previously failed when path contained any subset of its present value, OR-SEARCH returns failure. To avoid repeating sub-solutions we can label all new solutions found, record these labels, then return the label if these states are visited again. Post-processing can prune off unused labels. Alternatively, we can output a direct acyclic graph structure rather than a tree.