Question 1

3.13 Prove that GRAPH-SEARCH satisfies the graph separation property illustrated in Figure 3.9. (Hint: Begin by showing that the property holds at the start, then show that if it holds before an iteration of the algorithm, it holds afterwards.) Describe a search algorithm that violates the property

According to section 3.3 of the book graph separation property says that; the frontier separates the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier.

Let's take the start if the algorithm. In the start algorithm just has initial node which is the frontier itself. Hence any path must pass through the frontier as it is where the graph starts.

Now let's consider an iteration j of the graph search algorithm suggested in Fig 3.7 of the book. Here we have a frontier which has neither initial or goal state, nor is empty. From the frontier let's select a leaf node x. According to the algorithm at the end of the iteration x will be removed from the frontier and put in explored set, and its child will be placed in the frontier if they are not in the explored set.

Now let's consider two cases; one where path from explored to unexplored passes through x and another when it doesn't. For second case to reach unexplored set from explored set, by induction theory it should have at least one point from frontier at the beginning of iteration hence the property holds true.

For the first case, the next node x' must be a child of x and is not in the frontier. By description of the algorithm we know that every explored node is connected to the initial state by some combination of explored nodes. Hence x' cannot be in explored set as there is a path from x' to unexplored node not passing through the frontier. So, it will be added to the frontier and the path will now have a frontier node. This will make the property true as now every path in this case goes through x' which is in frontier.

The search algorithm which violates the property is the algorithm which ends up moving nodes from frontier to explored set before all child are generated. Another case occurs when some of the child nodes are not added to the frontier.

Question 2

3.16 A basic wooden railway set contains the pieces shown in Figure 3.32. The task is to connect these pieces into a railway that has no overlapping tracks and no loose ends where a train could run off onto the floor.

a. Suppose that the pieces fit together exactly with no slack. Give a precise formulation of the task as a search problem.

Initial state: Any of the above pieces in figure 3.32

Action: Joining one piece to another based on type

Next step generator (Transition function):

- For open piece add any other type

- For curved piece add in either orientation
- For a fork (switches and points) add in either direction connecting at either hole

Goal test: Check if all pieces are connected without overlapping. Also check if there are loose ends

Path cost: Assume cost of 1 for each piece.

b. Identify a suitable uninformed search algorithm for this task and explain your choice.

My choice would be to use depth search algorithm here. The branch factor is less compared to depth as we have only 4 pieces which could result in branching, hence it is safe to assume all solutions will be at similar depth. We could use depth limited search also here but since we know the exact depth we don't need to add extra step for checking it.

Since the space is very large breadth search or uniform cost search to work, as in this case owing to low branching factor depth first will save a lot of space complexity. Coming to iterative deepening it again does unnecessary work of checking depth every time. Hence Depth first is perhaps the best option.

c. Explain why removing any one of the "fork" pieces makes the problem unsolvable.

There are 4 fork pieces in the set 2 switches and 2 points, if a switch branches the way a point would join it back to one point. As we need no overlapping a or loose ends the only way to do that is to have same number switches and points or else somewhere there will be an open end. Hence removing just of the pieces would make the problem unsolvable.

d. Give an upper bound on the total size of the state space defined by your formulation. (Hint: think about the maximum branching factor for the construction process and the maximum depth, ignoring the problem of overlapping pieces and loose ends. Begin by pretending that every piece is unique.)

The maximum number of open holes would be 3 as we have just two switches and one any of the starting piece. Let's assume each piece is unique, hence we have total 12 (singles) + 2 (orientation) * 16 (curved) + 2 (orientation) * 2 (point) + 2 (orientation) * 2 (switches) * 2 (holes) = 56 choices.

The total number of pieces of wood is 32 so the max depth could be 32.

And since there can be at most 3 open holes and we have 56 choices, so it becomes 56*3 = 168.

So finally, the upper bound of state space becomes 168^32 / (12! *16! *2! *2!), as we are assuming each piece is unique.

Question 3

3.26 Consider the unbounded version of the regular 2D grid shown in Figure 3.9. The start state is at the origin, (0,0), and the goal state is at (x, y).

a. What is the branching factor b in this state space?

The branching factor for the figure is equal to the neighbors of each location which is equal to 4.

b. How many distinct states are there at depth k (for k> 0)?

The number of distinct state at depth k is 4k.

c. What is the maximum number of nodes expanded by breadth-first tree search?

BFS tree search will contain nodes which will be revisited as it does not maintain the list of explored nodes. Hence, we get maximum number of nodes as ((4^(|x|+|y|+1)-1)/3) -1.

d. What is the maximum number of nodes expanded by breadth-first graph search?

The BFS graph search algorithm will check for revisited state hence maximum no. of nodes will be:

2(|x|+|y|)(|x|+|y|+1)-1.

e. Is h = |u − x| + |v − y| an admissible heuristic for a state at (u, v)? Explain.
The heuristic h given in the question exactly denotes the Manhattan distance of the point from start state to the end state. Since it is a rectangular grid the shortest path length is always equal to the Manhattan distance, hence the heuristic is admissible.

f. How many nodes are expanded by A∗ graph search using h?

The A* algorithm would expand the nodes by |x| * |y|

g. Does h remain admissible if some links are removed?

If some links are removed the distance between the start and (x,y) will either increase or will remain same which means h is still admissible

h. Does h remain admissible if some links are added between nonadjacent states?

If links are added between nonadjacent states there will be some points whose distance will decrease from other points compared to Manhattan distance, hence h will not be admissible.

Question 4

4.3 In this exercise, we explore the use of local search methods to solve TSPs of the type defined in Exercise 3.30.

a. Implement and test a hill-climbing method to solve TSPs.

Distance matrix	Mumbai	Ahemdabad	Delhi	Chennai	Banglore
Mumbai	0	400	1200	1000	500
Ahemdabad	400	0	1000	1600	1200
Delhi	1200	1000	0	2000	1500
Chennai	1000	1600	2000	0	500
Banglore	500	1200	1500	500	0

```
Result
['Mumbai', 'Delhi', 'Chennai', 'Ahemdabad', 'Banglore'] 6000
['Chennai', 'Delhi', 'Banglore', 'Ahemdabad', 'Mumbai'] 5100
['Delhi', 'Chennai', 'Ahemdabad', 'Banglore', 'Mumbai'] 5300
['Delhi', 'Chennai', 'Ahemdabad', 'Banglore', 'Mumbai'] 5300
['Delhi', 'Chennai', 'Ahemdabad', 'Banglore', 'Mumbai'] 5300
['Banglore', 'Delhi', 'Chennai', 'Ahemdabad', 'Mumbai'] 5500
```

```
['Banglore', 'Delhi', 'Chennai', 'Ahemdabad', 'Mumbai'] 5500
['Banglore', 'Delhi', 'Chennai', 'Ahemdabad', 'Mumbai'] 5500
['Ahemdabad', 'Chennai', 'Delhi', 'Banglore', 'Mumbai'] 5600
['Chennai', 'Delhi', 'Banglore', 'Ahemdabad', 'Mumbai'] 5100
['Chennai', 'Delhi', 'Banglore', 'Ahemdabad', 'Mumbai'] 5100

 So the path with least distance is :  ['Chennai', 'Delhi', 'Banglore',
'Ahemdabad', 'Mumbai']
 The Distance to ravel for this path is  5100
```

b. Repeat part (a) using a genetic algorithm instead of hill climbing. You may want to consult Larra˜naga et al. (1999) for some suggestions for representations.

Question 5

4.5 The AND-OR-GRAPH-SEARCH algorithm in Figure 4.11 checks for repeated states only on the path from the root to the current state. Suppose that, in addition, the algorithm was to store every visited state and check against that list. (See BREADTH-FIRST-SEARCH in Figure 3.11 for an example.) Determine the information that should be stored and how the algorithm should use that information when a repeated state is found. (Hint: You will need to distinguish at least between states for which a successful subplan was constructed previously and states for which no subplan could be found.) Explain how to use labels, as defined in Section 4.3.3, to avoid having multiple copies of subplans.

Below table shows the modified algorithm. The main changes come in OR-SEARCH function. Firstly, it tries to recall the success of the state if it has been previously solved and it also checks if the state has failed before and if it contained any subset of present value it still returns failure. Not that we are not allowing cycles here.

If none of the conditions are true, then it checks for a new state and put it under the list of failure or success respectively. Hence it covers both part one where subplan exists previously and another where subplan doesn't exist.

So the extra information to store here is subplans which have been used before and whether that subpart has been successful or failure and we used this check to avoid pitfall of doing the same calculation again.

```
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 solved then return RECALL-RECORDED-SUCCESS(state)
    if state has previously been failed for a subset of path then return failure
    if state is on path then
        RECORD-FAILURE(state, path)
        return failure

    for each 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 a 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]
```