Pedram Safaei
CS 477
HW7

1.

a)

Kruskal's algorithm first sorts the edges by weight, then selects the edges that have the least weight, but that still add to the span of the tree that is forming without producing a cycle.

Another way is to start with a forrest of trees consisting of a solitary vertex each. Then add the edges (light to heavy) if the edge would connect two trees to form a larger tree.

Nth edge added	Edge	Weight	Keep
1	(g,h)	1	Yes
2	(A,c)	3	Yes
3	(h,i)	5	Yes
4	(A,b)	7	Yes
5	(c,d)	9	Yes
6	(c,g)	10	Yes
0	(g,i)	11	NO
0	(b,d)	12	NO
0	(d,i)	14	NO
7	(f,g)	15	Yes
8	(e,c)	17	Yes

b)

Prim's algorithm starts with an arbitrary vertex and then grows the spanning tree by adding the lightest edge that will grow the most current version of the spanning tree.

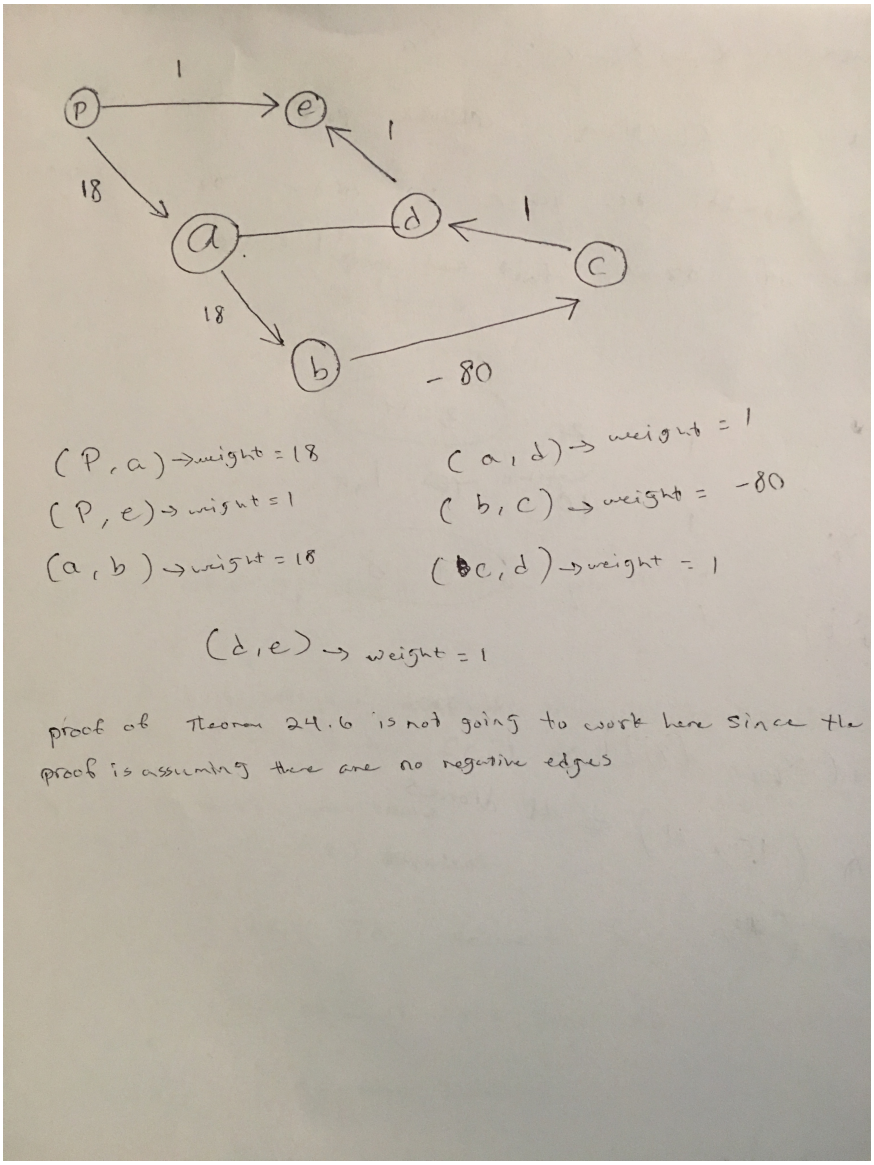
Nth edge added	Edge	Weight
1	(A,c)	3
2	(A,b)	7
3	(c,d)	9
4	(c,g)	10
5	(g,h)	1
6	(h,i)	5
7	(g,f)	15
8	(c,e)	17

2.
G = (V,E) → connected, undirected graph.

If we modify the depth first search algorithm we can identify a path that would traverse every edge in the graph twice. We know that the DFS already explores all edges in a graph, to track the order in which the edges are traversed we need to add a list to the algorithm that lists each vertex as it is seen or visited, once the algorithm ends the list will contain an order of traveling to each vertex hence crossing each edge twice. The DFS will be Θ (|V| + |E|). Now in the case of maze, we can consider all branches and termination of paths a vertex so every path would be an edge. As we go on the path we could leave a penny at the beginning and do the same as we are leaving at the end, if we reach to a dead end we drop a penny so we can mark it as visited and we go back to the last vertex and we place another penny at the beginning, and when we have a choice we always go with a path that has no penny and if there are 2 pennies in a path never take it, by doing this we will find the end without exploring the paths more than necessary.

3.
We need to identify the strongly connected components (SCCs) of G, then once the strongly connected components have been identified, we add enough arcs to make the future SCCs of G' be strongly connected, and then just enough arcs to connect the SCCs to one another to produce a component graph similar to that of G.
We need to take into account the fact that G' does not actually need to be constructed directly from the elements of G, it just has to have matching components; this means that we are actually free to create arcs as needed instead of selecting from the arc set of G, but a similar vertex set is required in order to have the same components.
We can start with the algorithm from the text that identifies the SCCs of a graph by performing two depth first searches, one on G and one on GT , SCC. This will take O |V | + |E|
Now we have various subsets of vertices that will constitute the SCCs that we need to match, to ensure that the SCCs are actually strongly connected, we need to add arcs, we can do this by making one simple cycle that runs through all of the vertices of an SCC. This is kind of a linked list structures of vertices, do this for all SCCs, meaning we add a total of V arcs (time → O |V |).
Finally, for each pair of SCCs, we create an arc to connect the pair in a way that matches the component graph of G. This will take O |E| in terms of the size of the arc set of the component graph of G.
Total running time = O |V | + |E| + O |V | + O |E| = O |V | + |E| .

4.
We know Djikstra's algorithm fails with negative eges since vertices are not revisited, and that will also affect the order we consider the vertices. One example would the the below algorithm w(p,e) < w(p,a) since e is addressed before a when we relax the edges and this happens again where d is considered before b when we are relaxing, so e can't be reached again for relaxation since p and d have been visited in the relaxation step. This will cause the algorithm to fail since the negative edge will be considered in the relaxation step too late in the execution.



5.
We know the weight of the shortest paths for all pairs, we can see if a vertex is an immediate predecessor to another on a shortest path. We Consider a pair of vertices i and j, where i is the source vertex and j is the destination. We can determine if another vertex k is a predecessor to j on that shortest path if the shortest weight of the path from k to j complements the path from i to k, that is, if the path from i to k and the final edge from k to j have the same sum as the shortest path from i to j, then k must be a valid predecessor to j on the shortest path.
Considering every pair of start and end vertices, and then checking every vertex to see if it is a predecessor on the shortest path between the pair, we can find the predecessors of every vertex on a shortest path.
Note that it is possible that there could be multiple such k vertices, since there could be multiple equivalent shortest paths from i to j; the algorithm needs to only store the last predecessor considered. The algorithm will need to produce a possibility for the predecessor matrix.
Since for each of n initial vertices we are checking n end vertices to form a pair, and we then check n intermediate vertices, this algorithm is O ($n * n * n$) = O (n^3).

```
Algorithm  FIND-PRED (L((1...n)(1...n)), W ((1...n)(1...n)))

    Create Π((1...n)(1...n))
    for  i ←1 to n
         do  for  j ←1 to n
                  do  π_{i,j}  = nil


    for   i ←1 to n
         do  for   j ←1 to n
                  do  for   k ←1 to n
                           do
                                if   l_{i,j}  = l_{ik}  + w(k, j)
                                   then  π_{i,j}  = k
```

6.
We know that probabilities are all independent, and all probabilities will have the largest product. We know that the probability that a channel will not fail will be its own probability multiplied with the largest probability of success up to that point.
And this problem is similar to shortest paths problem. We can use the same algorithms for finding a shortest path by replacing addition with multiplication of edge weights, and selecting the maximum result instead of the minimum.
Shortest path between a pair algorithms are not better than single source or single destination shortest path algorithms, we will not take that into account with the best path between only a particular pair, since we are not given any indication of which vertices/paths to consider, we just compute the best paths between all vertices.
Hence the task can be accomplished by modifying the Floyd-Warshall algorithm
$d^k_{i,j} = \max(d^{k-1}_{i,j}, d^{k-1}_{i,k} * d^{k-1}_{k,j})$
And this will give us
$\Theta (V^3)$