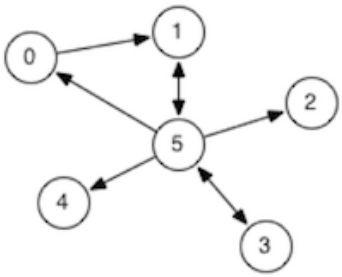
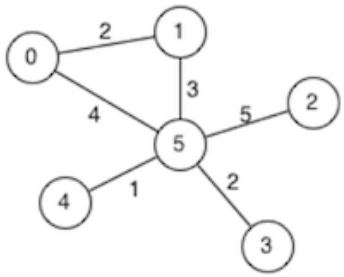
COMP9024 20T0 Data Structures and Algorithms

Week 3a Problem Set
Minimum Spanning Trees, Shortest Paths

1. (Graph representations)

   a. Consider the following graphs, where bi-directional edges are depicted as two-way arrows rather than having two separate edges going in opposite directions:





   For each of the graphs show the concrete data structures if the graph was implemented via:

   1. adjacency matrix representation (assume full V×V matrix)
   2. adjacency list representation (if non-directional, include both (v,w) and (w,v))

   b. Consider the following map of streets in the Sydney CBD:

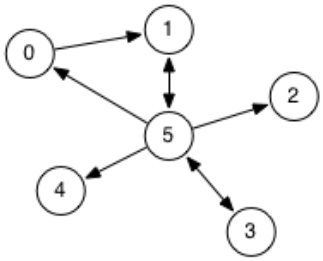
   Represent this as a directed graph, where intersections are vertices and the connecting streets are edges. Ensure that the directions on the edges correctly reflect any one-way streets (this is a driving map, not a walking map). You only need to make a graph which includes the intersections marked with red letters. Some things that don't show on the map: Castlereagh St is one-way heading south and Curtin Pl is a little laneway that you can't drive down.

   For each of the following pairs of intersections, indicate whether there is a path from the first to the second. Show a simple path if there is one. If there is more than one simple path, show two different paths.

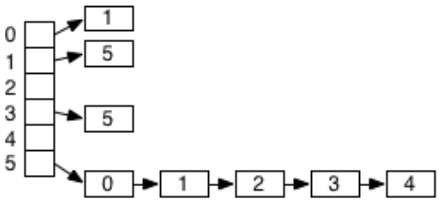
   1. from intersection "D" on Margaret St to insersection "L" on Pitt St
   2. from intersection "J" to the corner of Margaret St and York St (intersection "A")
   3. from intersection "P" on Castlereagh St to the corner of Margaret St and Wynyard Ln ("C")
   4. from the intersection of Castlereagh St and Hunter St ("M") to intersection "H" at Wynyard Park

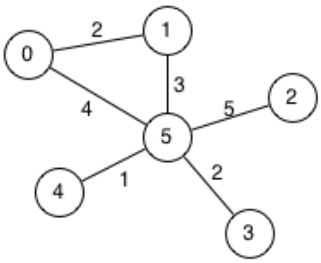
Answer:

a.

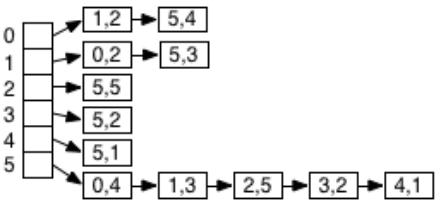


	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	0	0
3	0	0	0	0	0	1
4	0	0	0	0	0	0
5	1	1	1	1	1	0

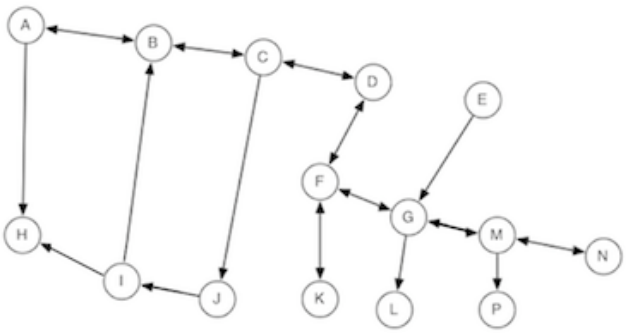




	0	1	2	3	4	5
0	0	2	0	0	0	4
1	2	0	0	0	0	3
2	0	0	0	0	0	5
3	0	0	0	0	0	2
4	0	0	0	0	0	1
5	4	3	5	2	1	0



b. The graph is as follows.

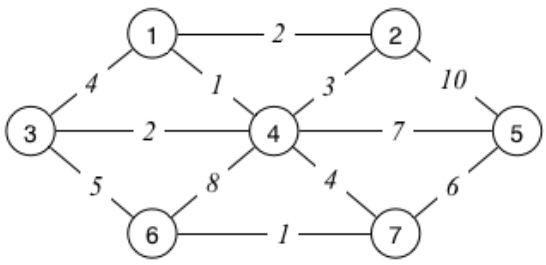


For the paths:

1. D → F → G → L and there are no other choices that don't involve loops through F or G.
2. J → I → B → A.
3. You can't reach C from P on this graph. Real-life is different, of course.
4. M → G → F → D → C → B → A → H. or M → G → F → D → C → J → I → H.

2. (Minimum spanning trees)

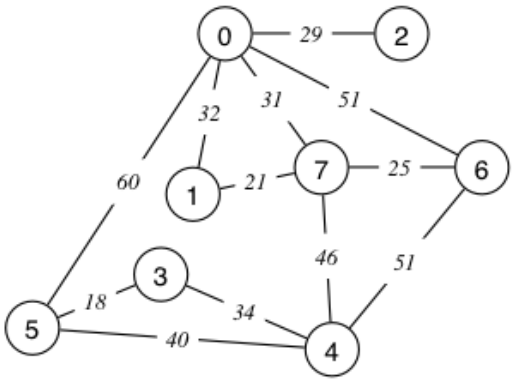
a. Show how Kruskal's algorithm would construct the MST for the following graph:



How many edges do you have to consider?

b. For a graph *G=(V,E)*, what is the least number of edges that might need to be considered by Kruskal's algorithm, and what is the most number of edges? Add one vertex and edge to the above graph to force Kruskal's algorithm to the worst case.

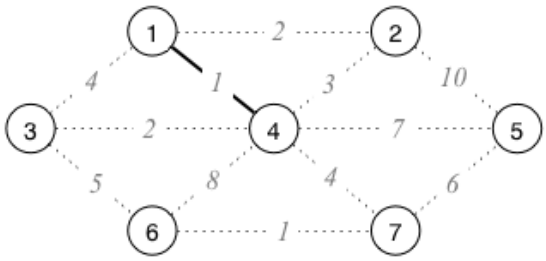
c. Trace the execution of Prim's algorithm to compute a minimum spanning tree on the following graph:



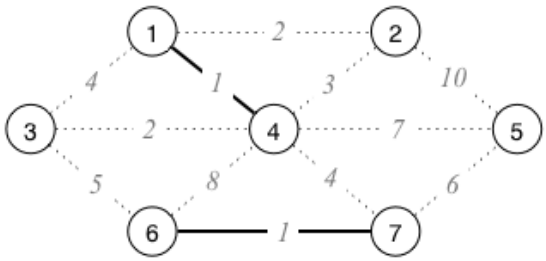
Choose a random vertex to start with. Draw the resulting minimum spanning tree.

Answer:

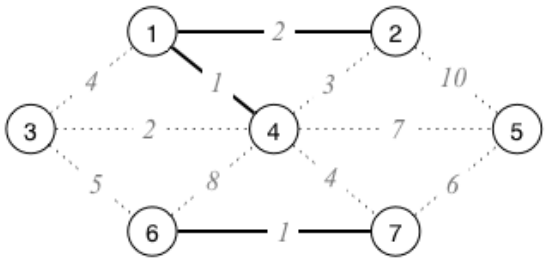
a. In the first iteration of Kruskal's algorithm, we could choose either 1-4 or 6-7, since both edges have weight 1. Assume we choose 1-4. Since its inclusion produces no cycles, we add it to the MST (non-existent edges are indicated by dotted lines):



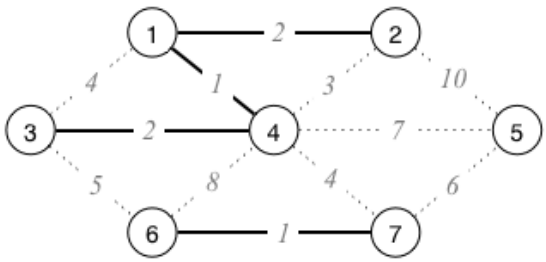
In the next iteration, we choose 6-7. Its inclusion produces no cycles, so we add it to the MST:



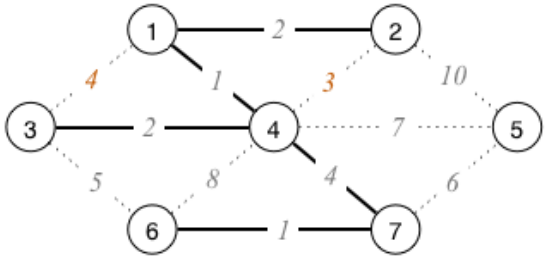
In the next iteration, we could choose either 1-2 or 3-4, since both edges have weight 2. Assume we choose 1-2. Since its inclusion produces no cycles, we add it to the MST:



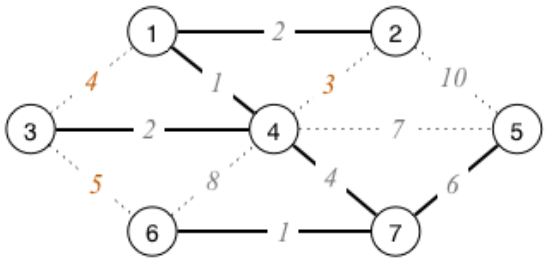
In the next iteration, we choose 3-4. Its inclusion produces no cycles, so we add it to the MST:



In the next iteration, we would first consider the lowest-cost unused edge. This is 2-4, but its inclusion would produce a cycle, so we ignore it. We then consider 1-3 and 4-7 which both have weight 4. If we choose 1-3, that produces a cycle so we ignore that edge. If we add 4-7 to the MST, there is no cycle and so we include it:



Now the lowest-cost unused edge is 3-6, but its inclusion would produce a cycle, so we ignore it. We then consider 5-7. If we add 5-7 to the MST, there is no cycle and so we include it:



At this stage, all vertices are connected and we have a MST.

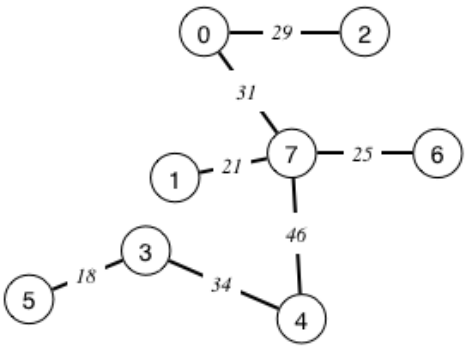
For this graph, we considered 9 of the 12 possible edges in determining the MST.

b. For a graph with *V* vertices and *E* edges, the best case would be when the first *V-1* edges we consider are the lowest cost edges and none of these edges leads to a cycle. The worst case would be when we had to consider all *E* edges. If we added a vertex 8 to the above graph, and connected it to vertex 5 with edge cost 11 (or any cost larger than all the other edge costs in the graph), we would need to consider all edges to construct the MST.

c. If we start at node 5, for example, edges would be found in the following order:

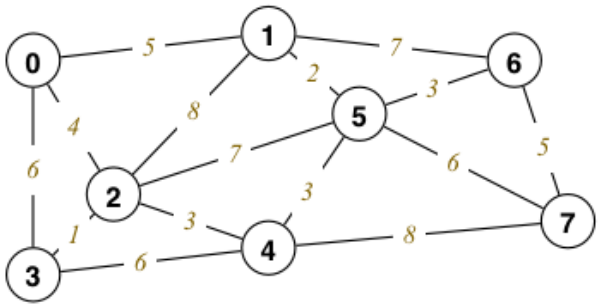
- 5-3
- 3-4
- 4-7
- 1-7
- 6-7
- 0-7
- 0-2

The minimum spanning tree:



3. (Dijkstra's algorithm)

a. Consider the following graph:



What is the shortest path from vertex 0 to vertex 3? From vertex 0 to vertex 7? How did you determine these?

b. Trace the execution of Dijkstra's algorithm on the graph from the previous question to compute the minimum distances from source node 0 to all other vertices.

Show the values of `vSet`, `dist[]` and `pred[]` after each iteration.

Answer:

a. Shortest path from vertex 0 to vertex 3 has total weight 5. There are really only two plausible choices: either the direct edge from vertex 0 to vertex 3, or two edges via vertex 2. It is easy to see that the total weight on the two-edge path (5) is less than the weight on the single edge (6).

Shortest path from vertex 0 to vertex 7 has total weight 13. Probably (although not necessarily) the best paths are also the most direct ones, which suggests 0-1-6-7, 0-1-5-7 and 0-2-4-7. If we sum the lengths on each of these paths, they are, respectively: 17, 13, 15. So we choose the shortest: 0-1-5-7.

b. Initialisation:

```
vSet = { 0, 1, 2, 3, 4, 5, 6, 7 }
dist = [ 0, ∞, ∞, ∞, ∞, ∞, ∞, ∞ ]
pred = [ -1, -1, -1, -1, -1, -1, -1, -1 ]
```

The vertex in vSet with minimum dist[] is 0. Relaxation along the edges (0,1,5), (0,2,4) and (0,3,6) results in:

```
vSet = { 1, 2, 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 6, ∞, ∞, ∞, ∞ ]
pred = [ -1, 0, 0, 0, -1, -1, -1, -1 ]
```

Now the vertex in vSet with minimum dist[] is 2. Considering all edges from 2 to nodes still in vSet:

- relaxation along (2,1,8) does not give us a shorter distance to node 1
- relaxation along (2,3,1) yields a smaller value (4+1 = 5) for dist[3], and pred[3] is updated to 2
- relaxation along (2,4,3) yields a smaller value (4+3 = 7) for dist[4], and pred[4] is updated to 2
- relaxation along (2,5,7) yields a smaller value (4+7 = 11) for dist[5], and pred[5] is updated to 2

```
vSet = { 1, 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 11, ∞, ∞ ]
pred = [ -1, 0, 0, 2, 2, 2, -1, -1 ]
```

Next, we could choose either 1 or 3, since both vertices have minimum distance 5. Suppose we choose 1. Relaxation along (1,5,2) and (1,6,7) results in new values for nodes 5 and 6:

```
vSet = { 3, 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, ∞ ]
pred = [ -1, 0, 0, 2, 2, 1, 1, -1 ]
```

Now we consider vertex 3. The only adjacent node still in vSet is 4, but there is no shorter path to 4 through 3. Hence no update to dist[] or pred[]:

```
vSet = { 4, 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, ∞ ]
pred = [ -1, 0, 0, 2, 2, 1, 1, -1 ]
```

Next we could choose either vertex 4 or 5. Suppose we choose 4. Edge (4,7,8) is the only one that leads to an update:

```
vSet = { 5, 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 12, 15 ]
pred = [ -1, 0, 0, 2, 2, 1, 1, 4 ]
```

Vertex 5 is next. Relaxation along edges (5,6,3) and (5,7,6) results in:

```
vSet = { 6, 7 }
dist = [ 0, 5, 4, 5, 7, 7, 10, 13 ]
pred = [ -1, 0, 0, 2, 2, 1, 5, 5 ]
```

Of the two vertices left in vSet, 6 has the shorter distance. Edge (6,7,5) does not update the values for node 7 since dist[7]=13<dist[6]+5=15. Hence:

```
vSet = { 7 }
dist = [ 0, 5, 4, 5, 7, 7, 10, 13 ]
pred = [ -1, 0, 0, 2, 2, 1, 5, 5 ]
```

Processing the last remaining vertex in vSet will obviously not change anything. The values in pred[] determine shortest paths to all nodes as follows:

```
0: distance = 0, shortest path: 0
1: distance = 5, shortest path: 0-1
2: distance = 4, shortest path: 0-2
3: distance = 5, shortest path: 0-2-3
4: distance = 7, shortest path: 0-2-4
5: distance = 7, shortest path: 0-1-5
6: distance = 10, shortest path: 0-1-5-6
7: distance = 13, shortest path: 0-1-5-7
```