Conrad Lewin
CS 325
5/6/2016

1) Queue:

Steps	**A | A.π**	**B | B.π**	**C | C.π**	**D | D.π**	**E | E.π**	**F | F.π**	**G | G.π**	**H | H.π**	**I | I.π**	**J | J.π**
0	0 | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL
1	empty	2 | A	∞ | NIL	3 | A	∞ | NIL	∞ | NIL	∞ | NIL	4 | A	∞ | NIL	1 | A
2	empty	2 | A	∞ | NIL	3 | A	∞ | NIL	∞ | NIL	7 | J	4 | A	6 | J	empty
3	empty	empty	∞ | NIL	3 | A	5 | B	∞ | NIL	7 | J	4 | A	6 | J	empty
4	empty	empty	8 | D	empty	4 | D	5 | D	7 | J	4 | A	6 | J	empty
5	empty	empty	8 | D	empty	4 | D	2 | H	7 | J	empty	5 | H	empty
6	empty	empty	4 | F	empty	4 | D	empty	7 | J	empty	5 | H	empty
7	empty	empty	empty	empty	4 | D	empty	7 | J	empty	5 | H	empty
8	empty	empty	empty	empty	empty	empty	7 | J	empty	5 | H	empty
9	empty	empty	empty	empty	empty	empty	7 | J	empty	empty	empty
10	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty

Step 0:

- Add all vertices in the graph, setting their keys to infinity and their predecessors to NIL.
- Set the source node's (A) key to 0.

Step 1:

- Extract the vertex from the queue with the smallest key. The vertex that meets this criteria is *A* insofar as $w(A, A) = 0$.
- Find all vertices adjacent to *A* still in the queue, that is *J*, *B*, *D* and *H*.
- *J*, *B*, *D* and *H* have weights $w(A, J) = 1$, $w(A, B) = 2$, $w(A, D) = 3$ and $w(A, H) = 4$. The weight of each vertex from *A* is less than its current key (infinity). Therefore, the predecessor of each vertex *J*, *B*, *D* and *H* is updated to point to *A* and their keys now reflect their respective weights from *A*.

Step 2:

- Extract the vertex from the queue with the smallest key. The vertex that meets this criteria is *J* insofar as $w(A, J) = 1$.
- Find all vertices adjacent to *J* still in the queue, that is *I* and *G*.
- *I* and *G* have weights $w(J, I) = 6$ and $w(J, G) = 7$. The weight of each vertex from *J* is less than its current key (infinity). Therefore, the predecessor of each vertex *I* and *G* is updated to point to *J* and their keys now reflect their respective weights from *J*.

Step 3:

- Extract the vertex from the queue with the smallest key. The vertex that meets this criteria is *B* insofar as $w(A, B) = 2$.
- Find all vertices adjacent to *B* still in the queue, that is *D* and *E*.
- *D* and *E* have weights $w(B, D) = 7$ and $w(B, E) = 5$. The key of vertex *E* is less than its current

key (infinity), while the key of vertex *D* from *B* is greater than it current key (3). Therefore, the predecessor of vertex *E* is updated to point to *B* and its key now reflects its respective weight from *B*, while the attributes of vertex *D* remain unchanged.

Step 4:

- Extract the vertex from the queue with the smallest key. The vertex that meets this criteria is *D* insofar as $w(A, D) = 3$.
- Find all vertices adjacent to *D* still in the queue, that is *F, C* and *E.*
- *F, C* and *E* have weights $w(D, F) = 5$, $w(D, C) = 8$ and $w(D, E) = 4$. The key of each vertex is less than its current key. Therefore, the predecessor of each vertex is updated to point to *D* and their keys now reflects their respective weights from *D*.

Step 5:

- Extract the vertex from the queue with the smallest key. Two vertices *H* and *E* meet this criteria insofar $w(A, H) = 4$ and $w(D, E) = 4$. We will choose *H*.
- Find all vertices adjacent to *H* still in the queue, that is *I* and *F.*
- *I* and *F* have weights $w(H, I) = 5$, $w(H, F) = 8$ and $w(D, E) = 2$. The key of each vertex is less than its current key. Therefore, the predecessor of each vertex is updated to point to *H* and their keys now reflects their respective weights from *H*.

Step 6:

- Extract the vertex from the queue with the smallest key. The vertex that meets this criteria is *F* insofar as $w(H, F) = 2$.
- Find all vertices adjacent to *F* still in the queue, that is *C.*
- *C* has a weight $w(F, C) = 4$ which is less than its current key (8). Therefore, the predecessor of vertex *C* is updated to point to *F* and its key now reflects its respective weight from *F.*

Step 7:

- Extract the vertex from the queue with the smallest key. Two vertices *C* and *E* meet this criteria insofar $w(F, C) = 4$ and $w(D, E) = 4$. We will choose *C*.
- There are no vertices adjacent to *C* still in the queue, so its attributes remain unchanged.

Step 8:

- Extract the vertex from the queue with the smallest key. The vertex that meets this criteria is *E* insofar as $w(D, E) = 4$.
- Find all vertices adjacent to *E* still in the queue, that is *G.*
- *G* has a weight $w(E, G) = 8$. The key of vertex *G* from *J* is less than its key from *E* to *G*, so its attributes remain unchanged.

Step 9:

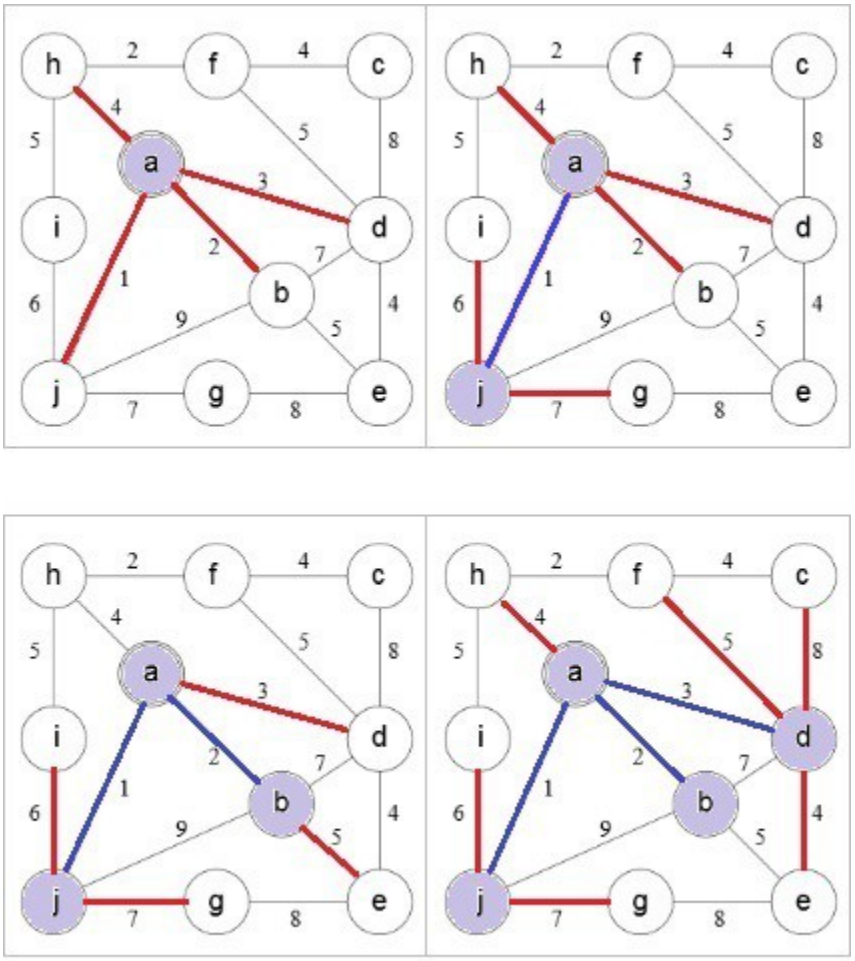
- Extract the vertex from the queue with the smallest key. The vertex that meets this criteria is *I* insofar as $w(H, I) = 5$.
- There are no vertices adjacent to *I* still in the queue, so its attributes remain unchanged.

Step 10:

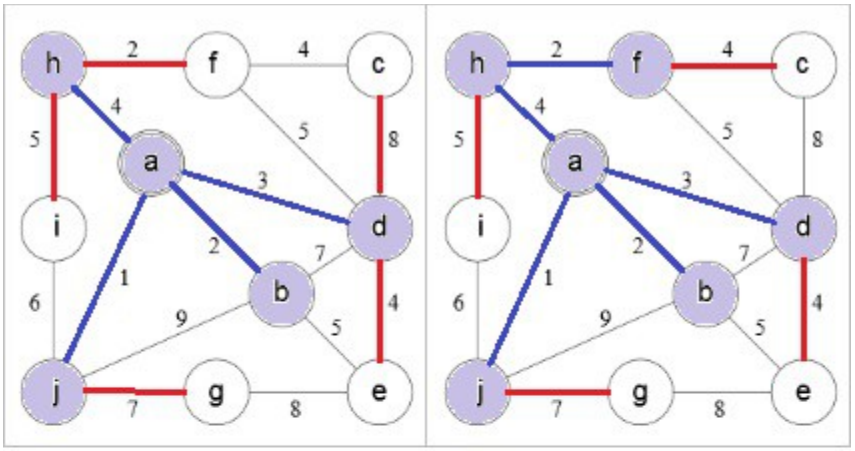
- Extract the vertex from the queue with the smallest key. The vertex that meets this criteria is *G* insofar as $w(J, G) = 7$.
- There are no vertices adjacent to *J* still in the queue, so its attributes remain unchanged.

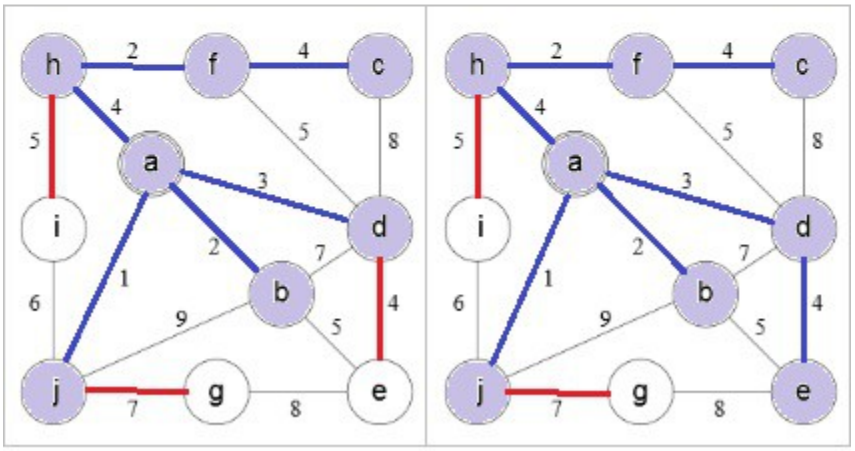
Below is an illustration of the steps listed above. White vertices are those still in the queue and grey vertices are those that have been removed from the queue and have been added to the minimum spanning tree. Red edges from gray vertices to white vertices indicate a predecessor relations, that is a white vertex has a gray predecessor. Blue edges are those that connect two gray vertices in the current minimum spanning tree.

Steps 1 – 4:

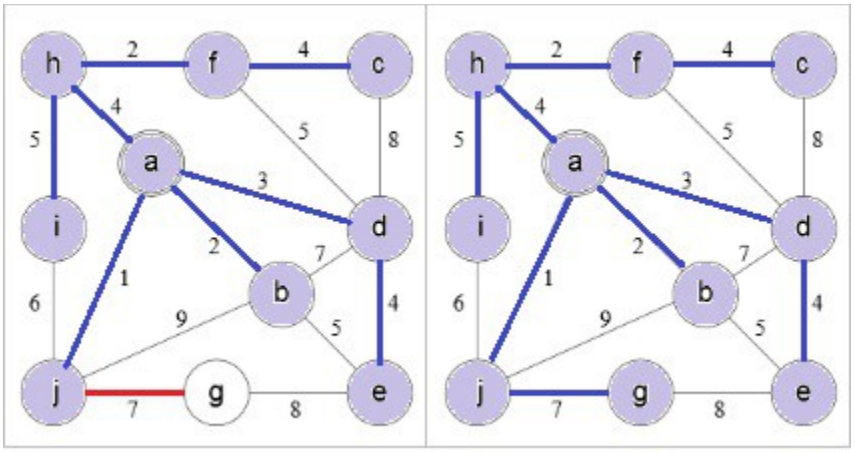


Steps 5 – 8:





Steps 9 – 10:



2) Priority Queue:

Steps	**A.d | A.π**	**B.d | B.π**	**C.d | C.π**	**D.d | D.π**	**E.d | E.π**	**F.d | F.π**	**G.d | G.π**	**H.d | H.π**	**S**
0	0 | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	empty
1	empty	2 | A	3 | A	6 | A	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	A
2	empty	empty	3 | A	6 | A	11 | B	∞ | NIL	∞ | NIL	∞ | NIL	A, B
3	empty	empty	empty	6 | A	11 | B	7 | C	∞ | NIL	∞ | NIL	A, B, C
4	empty	empty	empty	empty	11 | B	7 | C	∞ | NIL	∞ | NIL	A, B, C, D
5	empty	empty	empty	empty	8 | B	empty	∞ | NIL	15 | F	A, B, C, D, F
6	empty	empty	empty	empty	empty	empty	9 | E	15 | F	A, B, C, D, F, E
7	empty	empty	empty	empty	empty	empty	empty	15 | G	A, B, C, D, F, E, G
8	empty	empty	empty	empty	empty	empty	empty	empty	A, B, C, D, F, E, G, H

Step 0:

- Initialize the distance property of the source vertex to 0.
- Initialize the distance property of all other vertices to infinity.
- Initialize all vertex predecessors to NIL
- Initialize an empty set *S* of vertices that will represent all shortest paths in the graph.
- Add all vertices to the priority queue

Step 1:

- Extract the vertex with the smallest distance property from the priority queue, that is *A* insofar as *A*.d = 0.
- Add *A* to *S*.
- Find all vertices adjacent to *A* not already in *S*. These are *B, C* and *D*.
- Relax *B, D* and *C.* Since *A*.d + *w*(*A, B*) = 2 < *B*.d = ∞, *B*.d = 2; *A*.d + *w*(*A, C*) = 3 < *C*.d = ∞, *C*.d = 3 and *A*.d + *w*(*A, D*) = 6 < *D*.d = ∞, *D*.d = 6. The predecessor of each vertex is updated to point to *A*.

Step 2:

- Extract the vertex with the smallest distance property from the priority queue, that is *B* insofar as *B*.d = 2.
- Add *B* to *S*.
- Find all vertices adjacent to *B* not already in *S*. These are *D* and *E*.
- Relax *D* and *E.* Since *B*.d + *w*(*B, D*) = 9 > *D*.d = 6, *D*.d = 6 and *B*.d + *w*(*B, E*) = 11 < *E*.d = ∞, *E*.d = 11. The predecessor of vertex *E* is updated to point to *B*, while vertex *D* remains unchanged.

Step 3:

- Extract the vertex with the smallest distance property from the priority queue, that is *C* insofar as *C*.d = 3.
- Add *C* to *S*.

- Find all vertices adjacent to *C* not already in *S*, that is *F*.
- Relax *F*. Since *C*.d + *w*(*C*, *F*) = 7 < *F*.d = ∞, *F*.d = 7. The predecessor of vertex *F* is updated to point to *C*.

Step 4:

- Extract the vertex with the smallest distance property from the priority queue, that is *D* insofar as *D*.d = 6.
- Add *D* to *S*.
- Find all vertices adjacent to *F* not already in *S*, that is *F*.
- Relax *F*. Since *D*.d + w(*D*, *F*) = 9 > *F*.d = 7, *F*.d = 7. The predecessor of vertex *F* remains unchanged.

Step 5:

- Extract the vertex with the smallest distance property from the priority queue, that is *F* insofar as *F*.d = 7.
- Add *F* to *S*.
- Find all vertices adjacent to *F* not already in *S*, that is *E* and *H*.
- Relax *E* and *H*. Since *F*.d + w(*F*, *E*) = 8 < *E*.d = 11, *E*.d = 8 and *F*.d + w(*F*, *H*) = 15 < *H*.d = ∞, *H*.d = 15. The predecessor of each vertex is updated to point to *F*.

Step 6:

- Extract the vertex with the smallest distance property from the priority queue, that is *E* insofar as *E*.d = 8.
- Add *E* to *S*.
- Find all vertices adjacent to *E* not already in *S*, that is *G*.
- Relax *G*. Since *E*.d + w(*E*, *G*) = 9 < *G*.d = ∞, *G*.d = 9. The predecessor of vertex *G* is updated to point to *E*.

Step 7:

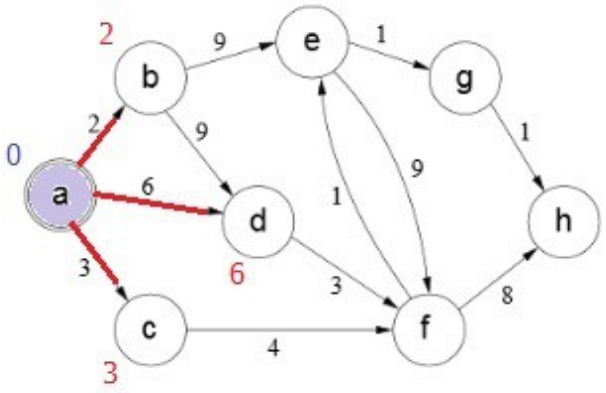
- Extract the vertex with the smallest distance property from the priority queue, that is *G* insofar as *G*.d = 9.
- Add *G* to *S*.
- Find all vertices adjacent to *G* not already in *S*, that is *H*.
- Relax *H*. Since *G*.d + w(*G*, *H*) = 10 < *H*.d = 15, *H*.d = 10. The predecessor of vertex *H* is updated to point to *G*.

Step 8:

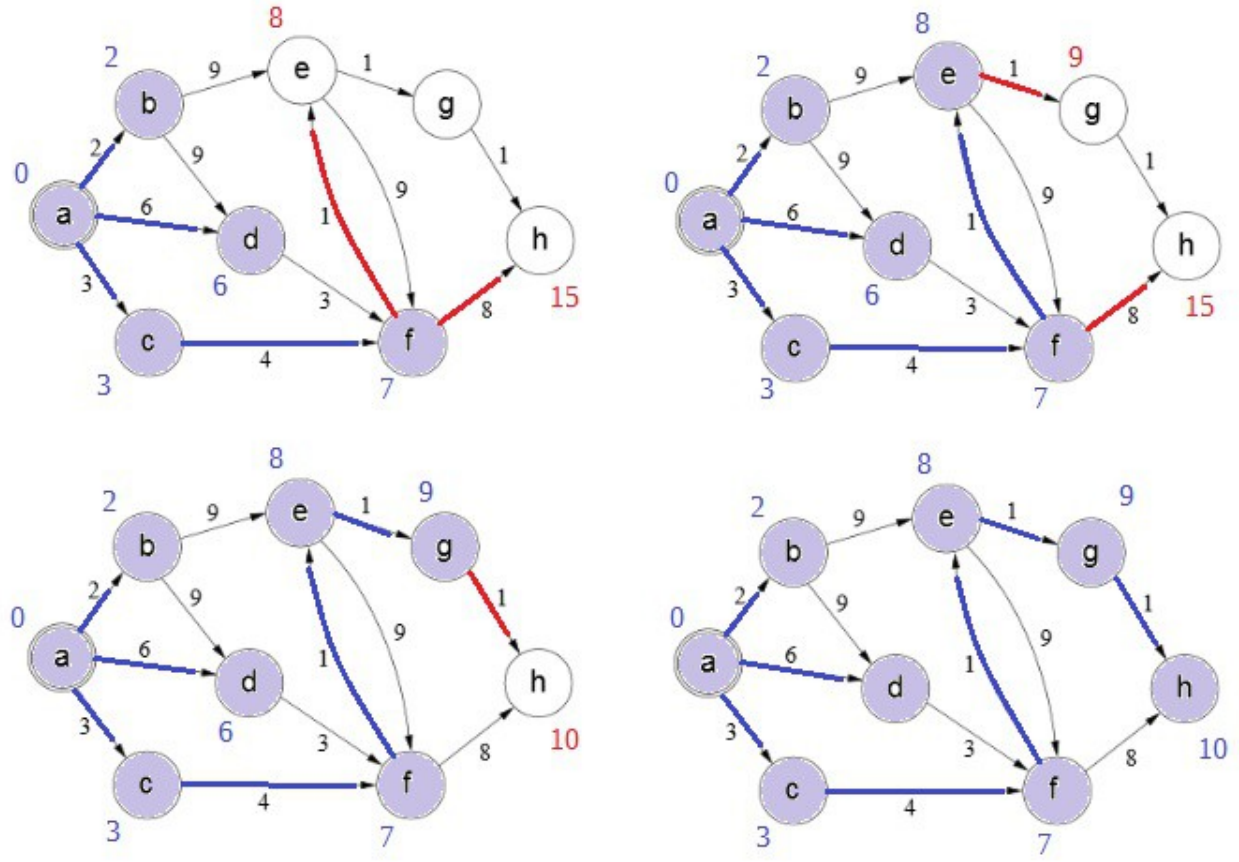
- Extract the vertex with the smallest distance property from the priority queue, that is *H* insofar as *H*.d = 10.
- Add *H* to *S*.
- Find all vertices adjacent to *H* not already in *S*. There are no vertices that meet this criteria.

Below is an illustration of the steps listed above. Red numbers indicate the current distance property of a white vertex, while blue numbers indicate the shortest path weight from *A* to the related gray vertex. Red edges represent predecessor relationships between white and gray vertices, while blue edges show shortest paths between gray vertices.

Steps 1 – 4:


0
2
3
6
11
a
b
c
d
e
f
g
h
2
9
1
6
3
4
8

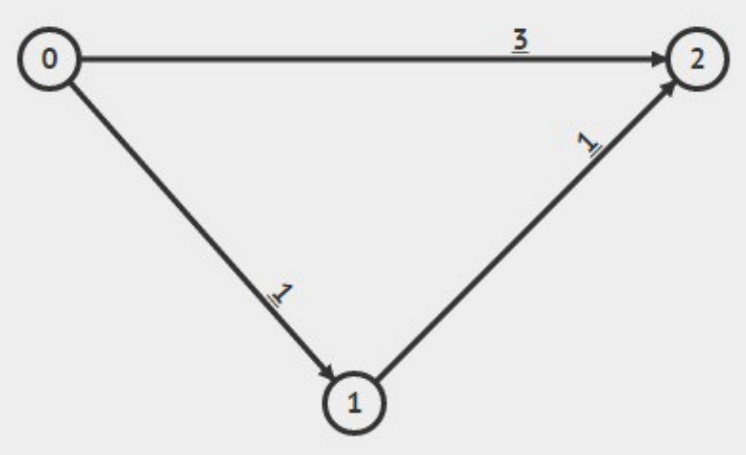
Steps 5 to 8:



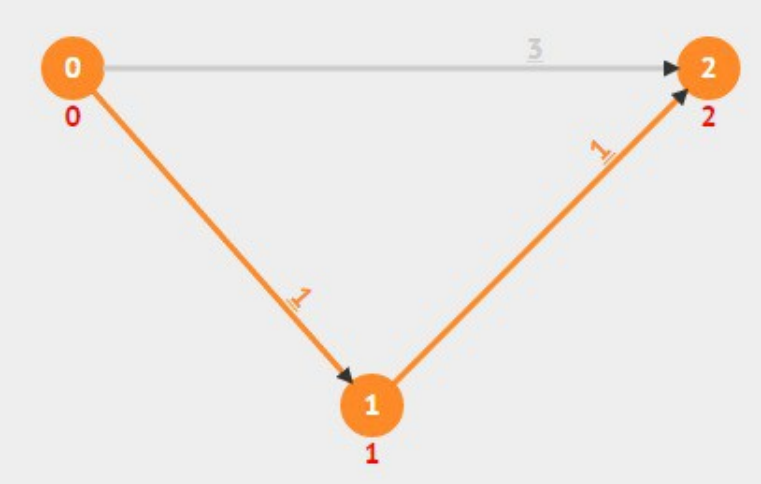
3.a) Consider a graph $G = (V, E)$ with nonnegative edge weights. Let T be the set of vertices that comprise some minimum spanning tree in a subgraph of G, and let $A = V - T$, that is those vertices in G that are not in the minimum spanning tree. If we cut the graph along those edges who have one endpoint in both T and another in A while respecting those edges that comprise the minimum spanning tree, using Prim's algorithm, we need only find a light edge crossing this cut to expand the minimum spanning tree.

Let u be some vertex in T and v_1 and v_2 be some vertices in A whose predecessor is u and whose edges cross the cut. Let $x = w(u, v_1)$ and $y = w(u, v_2)$. Furthermore, let $x \leq y$ so that (u, v_1) is a light edge and $T = T \cup \{v_1\}$. Now, let $x` = w(u, v_1) + 1$ and $y` = w(u, v_2) + 1$. This does not change the direction of the original inequality since $x` \leq y`$. Thus (u, v_1) is a still a light edge connecting T to A and T is still $T = T \cup \{v_1\}$. Therefore, uniformly increasing the weights of all the edges in E by 1 will not affect the minimum spanning tree.

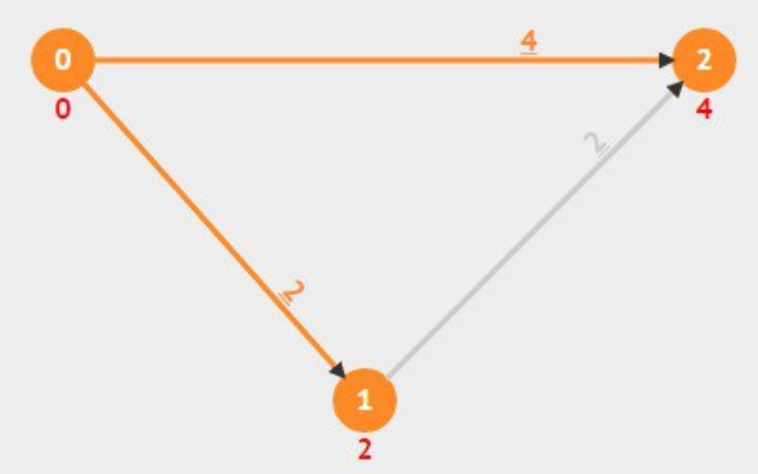
3.b) Consider the following graph:



Using Dijkstra's shortest path algorithm, we see that the shortest path from vertex 0 to all other vertices in the graph is:

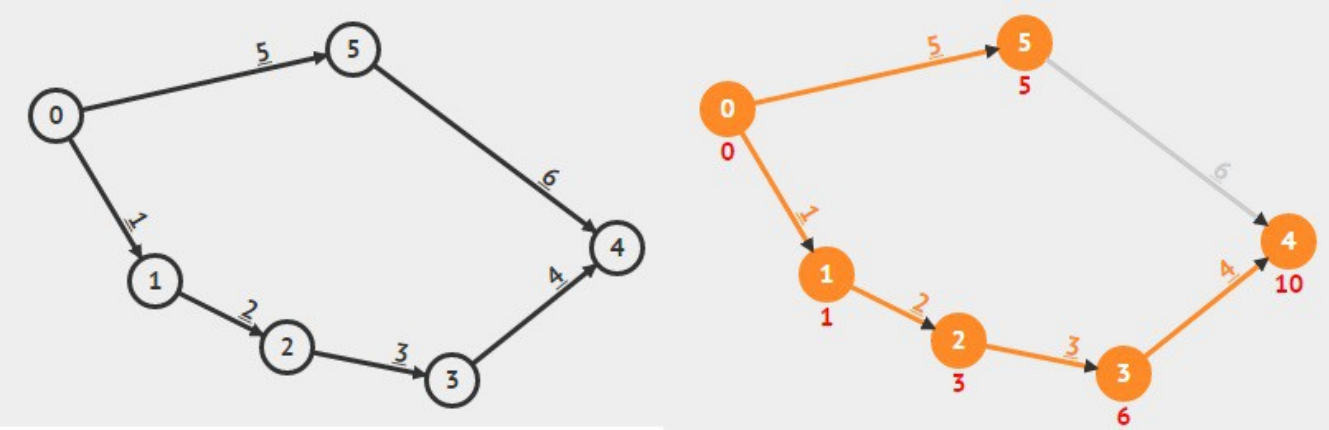


However, if we increase the weight of each edge by 1, this path will change:

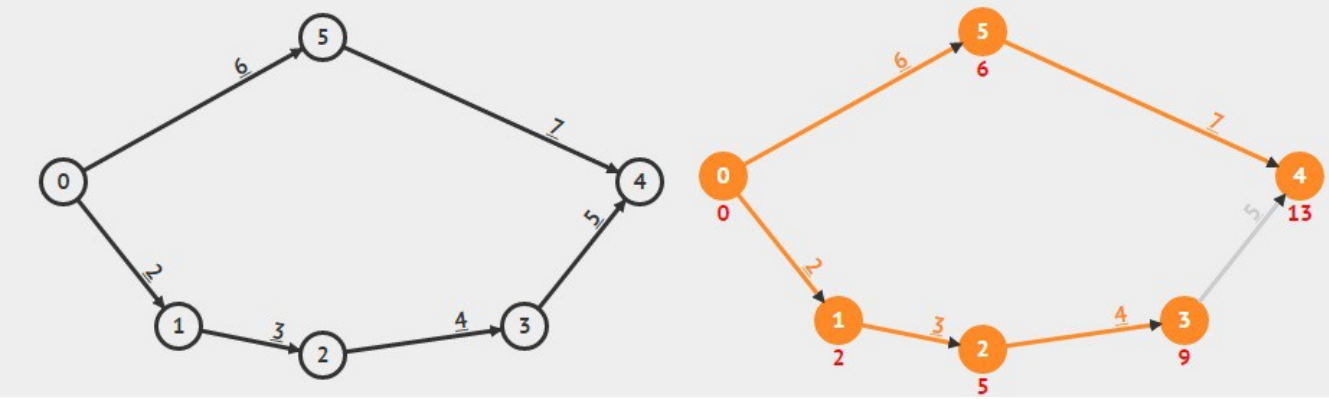


This occurs, because Dijkstra's algorithm sets the distance property of vertex 2 before it calculates the equivalent distance from vertex 1 to 2. Thus, when the path from vertex 1 to vertex 2 is considered, the algorithm finds no need to relax vertex 2's current distance property since 1.d + $w(1,2)$ = 2.d.

Another example occurs when there are many low weight edges in a shortest path s to v and some other path from s to v with few edges but larger edge weights. For example, consider the following graph and the shortest path from vertex 0 to 4:



If we increase the weights of all edges by 1, then the shortest path from vertex 0 to 4 changes from [0, 1, 2, 3, 4] in which 4.d = 14 to [0, 5, 4] in which 4.d = 13.



4.a)
A: discovery time = 1, finishing time = 12
C: discovery time = 2, finishing time = 7
E: discovery time = 3, finishing time = 6
F: discovery time = 4, finishing time = 5
B: discovery time = 8, finishing time = 11
D: discovery time = 9, finishing time = 10

4.b) Courses could be taken in the following order while maintaining all necessary prerequisites: A, B, D, C, E, F.

5) Pseudocode:

```
FindHamiltonianPath(G)
    path = TopologicalSort(G)
    predecessor = NIL
    for each u in path
        if  u.π != predecessor
            return false
        predecessor = u
    return true

//TopologicalSort() and Visit() are based off of CLRS pages 604 and 613
TopologicalSort(G)
    list = {}
    for each u ∈G.V
        u.color = white
        u.π = NIL
    time = 0

    for each vertex u ∈G.V
        if u.color == white
            Visit(G, u, list)

    return list

Visit(G, u, list)
    time = time + 1
    u.discovered = time
    u.color = gray
    for each v ∈G.Adj[u]
        if v.color == white
            v. π = u
            Visit(G, v, list)
        else
            return //DAG detected
    u.color = black
    time = time + 1
    u.finished = time
    AddFront(list, u)
```

The FindHamiltonianPath() function performs a depth-first search on a given graph, recording vertex predecessors, discovery and finished processing times. Once a vertex has been completely processed, it is added to the front of a linked list or array. After the helper function TopologicalSort() has completed, FindHamiltonianPath() should have a sorted list of vertices. The function will then iterate through this list, examining the predecessor of each vertex therein. If the current vertex's predecessor matches the vertex previously examined, that is, the current vertex is not adjacent to the previous one, then there is no Hamiltonian path in this graph. Otherwise, if each vertex is adjacent to the one that follows it in the sort, then there is a Hamiltonian path.

This works because a topological sort returns a list of vertices that directionally depend on one another insofar as one vertex cannot be visited unless its predecessor was traversed prior. Since a simple

Hamiltonian path must traverse each vertex exactly once, there can be no cycles used for backtracking and each edge must lead to successive vertices in a set order, namely the topological. By ensuring that there are no gaps in adjacency, and thus, no dead end forks in the tree that are unreachable by the path, we can verify that the topological sorting of the graphs vertices do in fact form a Hamiltonian path and is not just an ordering of dependencies.

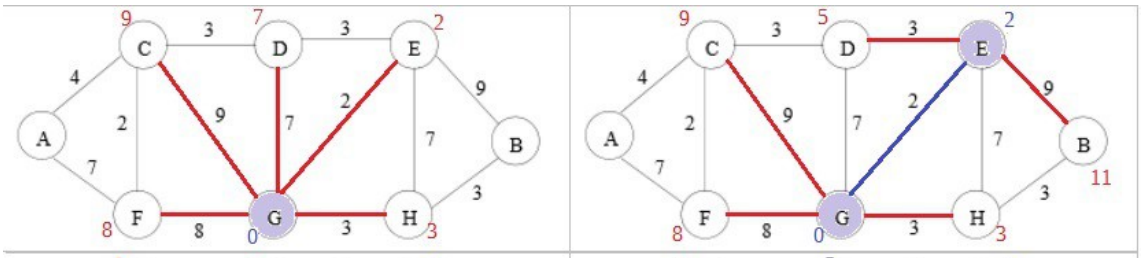
The run time for TopologicalSort() and its helper Visit() is equivalent to the run time of a standard DFS algorithm – $O(V + E)$. Note that adding an element to the front of a linked list (AddFront()) can be done in constant time. Thus, the only factor we must consider is the for loop in FindHamiltonianPath() that checks the predecessor of each vertex. In the worst case scenario, this will be done V times, meaning that the algorithm is $O(V + E) + V = O(V + E)$.

6.a) Dijkstra's algorithm can be used to find the shortest path from vertex *G* to all other vertices in the graph.

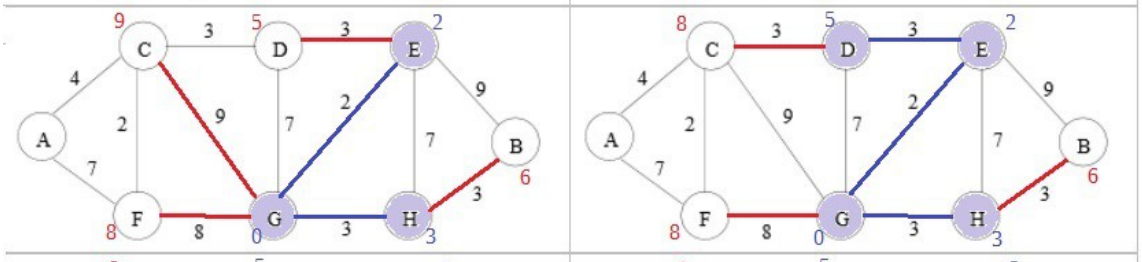
Priority Queue:

Steps	**A.d | A.π**	**B.d | B.π**	**C.d | C.π**	**D.d | D.π**	**E.d | E.π**	**F.d | F.π**	**G.d | G.π**	**H.d | H.π**	**S**
0	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	∞ | NIL	0 | NIL	∞ | NIL	empty
1	∞ | NIL	∞ | NIL	9 | G	7 | G	2 | G	8 | G	empty	3 | G	G
2	∞ | NIL	11 | E	9 | G	5 | E	empty	8 | G	empty	3 | G	G, E
3	∞ | NIL	6 | H	9 | G	5 | E	empty	8 | G	empty	empty	G, E, H
4	∞ | NIL	6 | H	8 | D	empty	empty	8 | G	empty	empty	G, E, H, D
5	∞ | NIL	empty	8 | D	empty	empty	8 | G	empty	empty	G, E, H, D, B
6	15 | F	empty	8 | D	empty	empty	empty	empty	empty	G, E, H, D, B, F
7	12 | C	empty	empty	empty	empty	empty	empty	empty	G, E, H, D, B, F, C
8	empty	empty	empty	empty	empty	empty	empty	empty	G, E, H, D, B, F, C

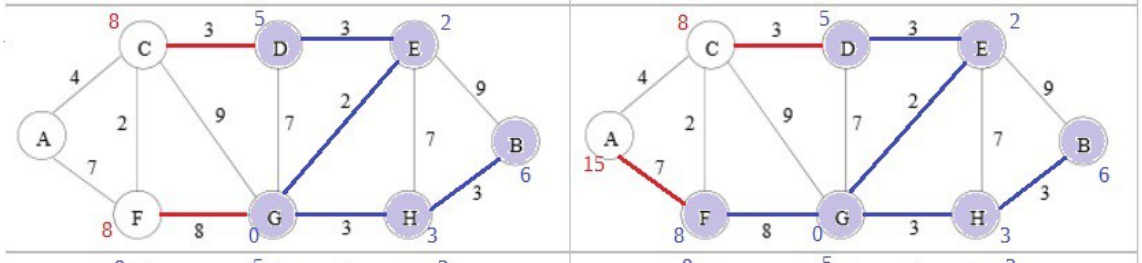
Steps 1 - 2



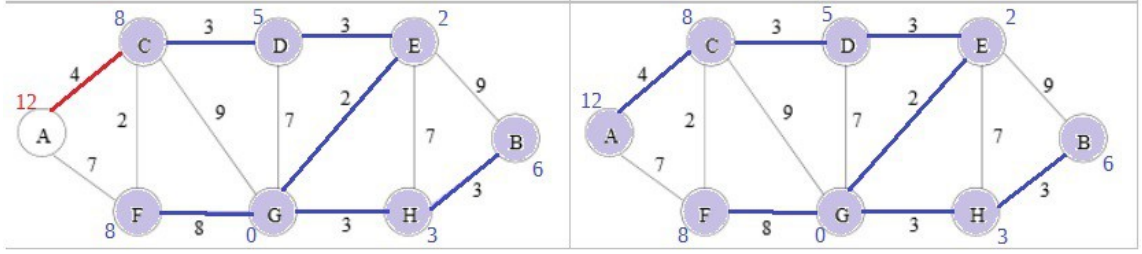
Steps 3 - 4



Steps 5 - 6



Steps 7 - 8

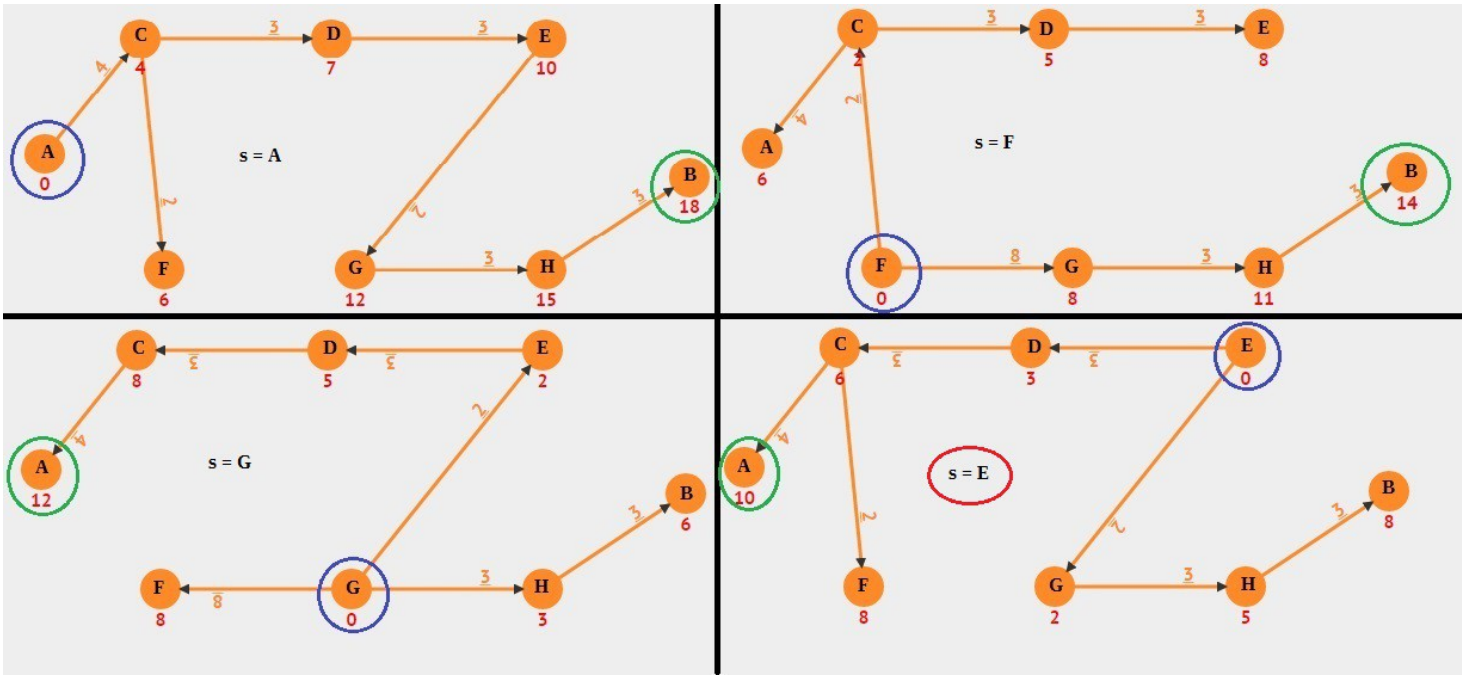


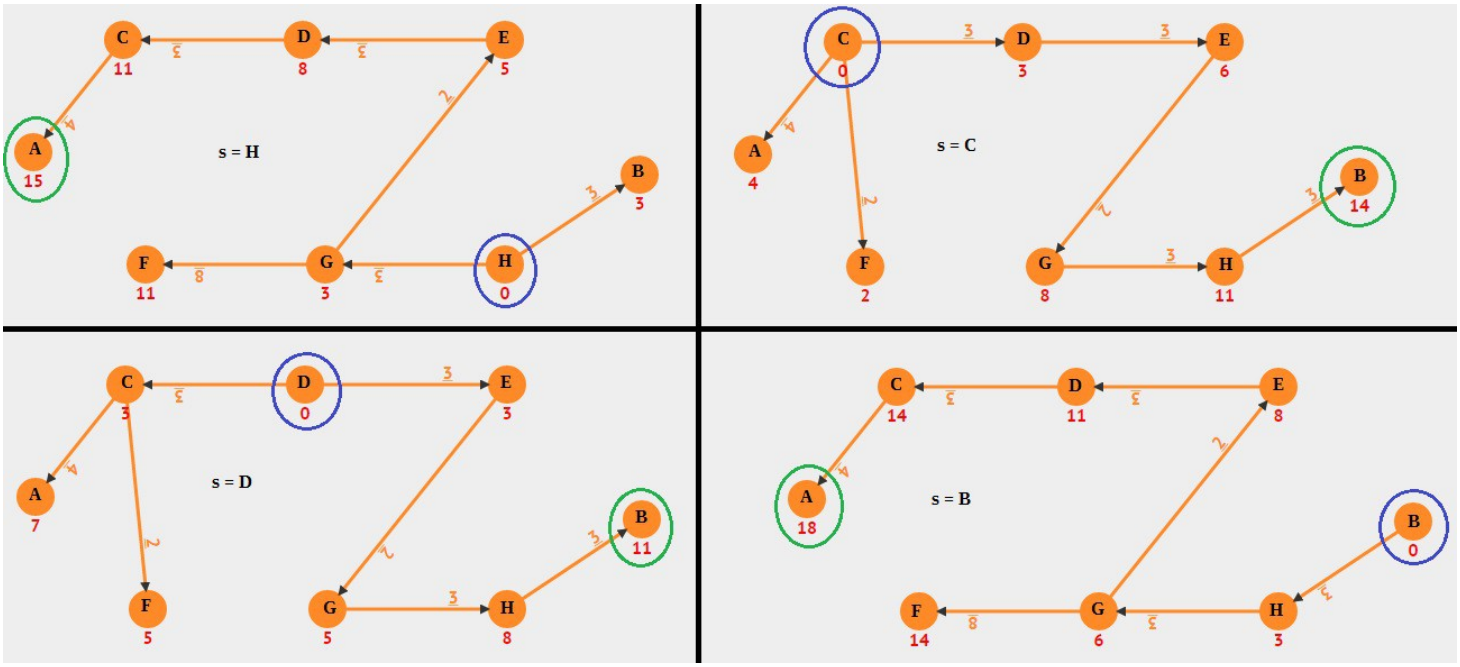
6.b) Considering a graph symbolizing a road map in which vertices *f* and edges *r* represent all intersections and roads respectively, we can find the optimal placement of the fire station at some vertex by running Dijkstra's algorithm on each vertex in the graph and tracking the longest path in each resulting tree. We start by adding *f* to a stack *S*, then we pop off the first element and run Dijkstra's algorithm on this vertex *s*. Ask the algorithm computes the shortest distance from *s* to each other vertex, we keep track of the longest path computed by comparing the previous path length to the current and keeping the max. At the completion of Dijkstra's algorithm, we should not only have the shortest path from *s* to all vertices in *f*, but also the largest distance *L* from *s* to some vertex f_i. At this point, we compare *L* with some local maximum. After the first iteration, we should assert that if $L < \infty$ then our new maximum is *L*. Subsequent comparisons will test the current value of *L* to this updated max value. After the last vertex has been processed, *L* should contain the shortest of the longest paths discovered in each shortest path solution. The vertex *s* associated with this *L* values will be the optimal placement for the fire station.

Dijkstra's algorithm can run in O(*f* lg f + *r*) time, assuming the use of a Fibonacci heap to implement the priority queue, populating the stack will take *f* constant time steps and both the comparison as well as the assignment of each *L* value can be done in constant time (in the worst case, this will happen *f* times). Since the algorithm described above must run Dijkstra's algorithm *f* times, we get a run time of

$f(f \lg f + r) + 2f = O(f^2 \lg f + fr)$.

6.c) Given the graph outlined in 6.a, the optimal placement for the fire station will be at vertex *E*. This is because the distance from *E* to that vertex furthest from it, in this case *A*, is the shortest of the longest paths in each shortest path tree calculated from each vertex in the graph. Shown below are depictions of the shortest paths using each vertex in the graph as a source. A blue circle marks the source vertex, a green circle marks that vertex furthest from the source and the red circle indicates the source vertex that serves as an optimal intersection in which to place the fire station.





Extra Credit:

We can use an algorithm similar to the one described in 6.b, having to make only a minor modification as to how the L values are dealt with. Rather than comparing the L values in order to find a minimum, each L value computed, along with its related source vertex, will be stored in a min heap. This can be done in constant time using a Fibonacci heap. Once each vertex has been processed, we extract two minimum elements from the heap. These elements will have the shortest longest paths of any source vertex in the given graph, so we can place a fire station at each of these intersections to produce an optimal solution. Inserting elements and extracting a minimum element takes constant time, however f elements will have to be inserted into the heap, while only 2 constant time operations will occur during the extraction phase. Therefore, the run time of this algorithm is roughly equivalent to the one described in 6.b. That is, $f(f \lg f + r) + 2f + 2 = O(f^2 \lg f + fr)$.