More Graph Algorithms - Minimum Spanning Trees

Definition: A spanning tree of a graph G is a tree (acyclic) that connects all of the vertices of G once. It "spans" the graph G.

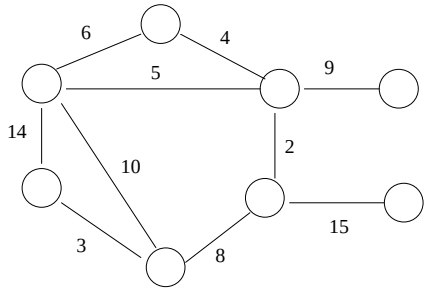
A minimum spanning tree is a spanning tree on a weighted graph that has the minimum total weight:

$w(T) = \sum_{u,v \in T} w(u,v)$ such that w(T) is minimum

i.e. the sum of all the edges in the tree is minimum. (Where might this be useful? Can be used to approximate some NP-Complete problems)

We will assume that the graph G is connected and undirected.

Example of MST:



(Shade in links to make this a MST)

Note: MST Problem has "Optimal Substructure." This essentially means that a subtree of the MST must in turn be a MST of the nodes that composes it. We will use the idea of optimal substructure in dynamic programming.

Claim: Say remove any edge (u,v) in a MST, e.g., edge with weight "2" in above tree. This results in two trees, T1 and T2. T1 is a MST of its subgraph, while T2 is a MST of its subgraph. Then the MST of T is T1+T2+edge "2". This edge is the smallest linking the two together, and there can't be a better tree than T1 or T2 or T would be suboptimal.

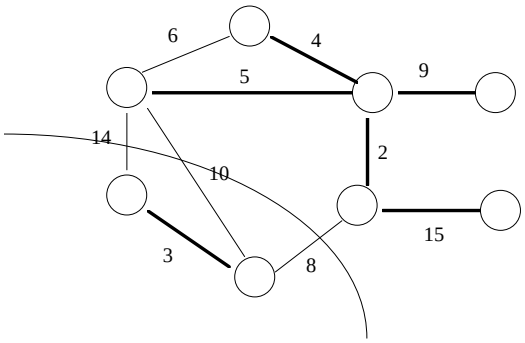
Algorithm to find MST: Greedy Algorithm

At each step a greedy algorithm will always make a choice that is best at that moment. For problems in general, sometimes this works but sometimes it does not! It happens to work for finding the MST.

Concept of the *cut*:

A cut of a graph G is a partition of G into two sets, S and V-S. The cut *crosses* an edge only once, but does not have to go through all edges. A cut *respects* an edge if it does not cross that edge. It turns out that for a MST, "safe" edges to add are those minimum edges that cross a cut that respects the edges we have included in our MST.

Sample Cut: Respects edges we have built up so far in our MST. Prevents adding edge to form a loop.



Idea: Greedy MST: Go through the list of edges and make a forest that is a MST

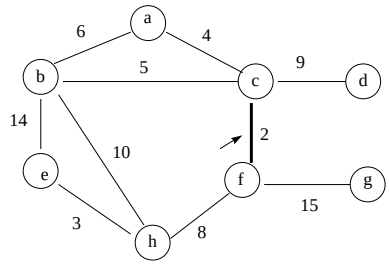
1. At each vertex, sort the edges
2. Edges with smallest weights examined and possibly added to MST before edges with higher weights
3. Edges added must be "safe edges" that do not ruin the tree property.

Kruskal's MST Algorithm:

```
Kruskal(G,w)              ; Graph G, with weights w
    A ← {}                ; Our MST starts empty
    for each vertex v ∈ V[G] do Make-Set(v)    ; Make each vertex a set
    Sort edges of E by increasing weight
    for each edge (u,v) ∈ E in order
        ;  Find-Set returns a representative (first vertex) in the set
        do if Find-Set(u) ≠ Find-Set(v)
           then A ← A ∪ {(u,v)}
                Union(u,v)                    ; Combines two trees
    return A
```

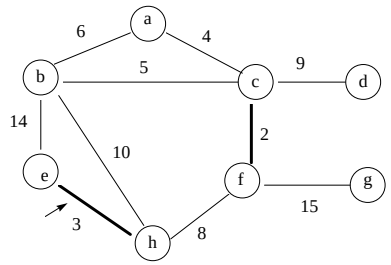
Example:

A={}, Make each element its own set. {a} {b} {c} {d} {e} {f} {g} {h}
Look at smallest edge first: {c} and {f} not in same set, add it to A, union together.



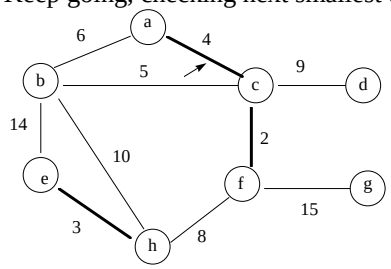
Now get {a} {b} {c f} {d} {e} {g} {h}

Keep going, checking next smallest edge. {e} <> {h}, add edge.



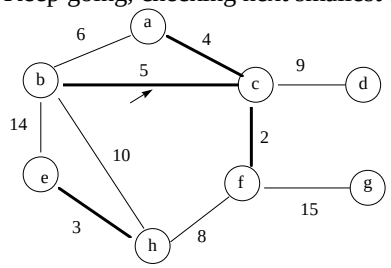
Now get {a} {b} {c f} {d} {e h} {g}

Keep going, checking next smallest edge. {a} <> {c f}, add edge.



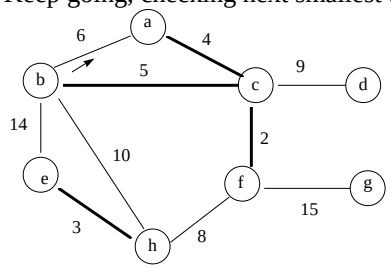
Now get {b} {a c f} {d} {e h} {g}

Keep going, checking next smallest edge. {b} <> {a c f}, add edge.

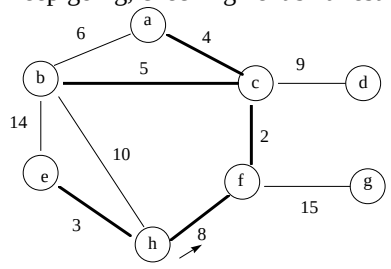


Now get {a b c f} {d} {e h} {g}

Keep going, checking next smallest edge. {a b c f} = {a b c f}, dont add it!

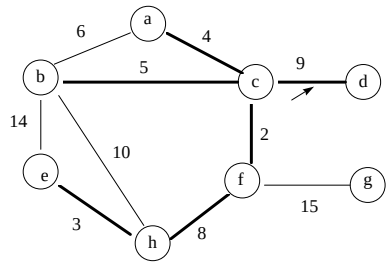


Keep going, checking next smallest edge. {a b c f} = {e h}, add it.



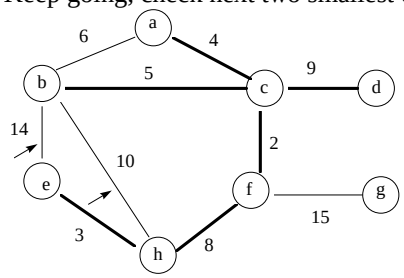
Now get {a b c f e h} {d}{g}

Keep going, checking next smallest edge. {d} <> {a b c e f h}, add it.

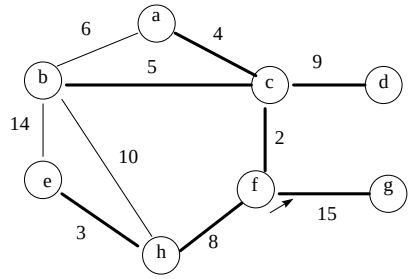


Now get {a b c d e f h} {g}

Keep going, check next two smallest edges. {a b c d e f h} = {a b c d e f h}, don't add it.



Do add the last one:



Implementation: Union, Make-Set, Find-Set

Array member[] : member[I] is a number j such that the ith vertex is a member of the jth set.
Example: member[1,4,1,2,2] indicates the sets S1={1,3}, S2={4,5} and S4={2}; i.e. position in the array gives the set number. Idea similar to counting sort, up to number of edge members.

Given this structure:

Make-Set(v)
member[v] ← v

Make-Set runs in constant running time for a single set.

Find-Set(v)
Return the first element of member[v]

Find-Set runs in constant time.

```
Union(u,v)
        for I=1 to n
                do if member[I] = u then member[I]=v
```

Scan through the member array and update old members to be the new set.
Running time O(n), length of member array.

Time for MST-Kruskal:

Initial Loop to Make-Set O(V)
Sorting the edges takes time O(ElgE) using heapsort or mergesort
Loop for each edge:
- Find-Set constant time
- Adding single edge to A constant time
- Union time O(V) in our algorithm (book has O(lgV) version using trickier methods)

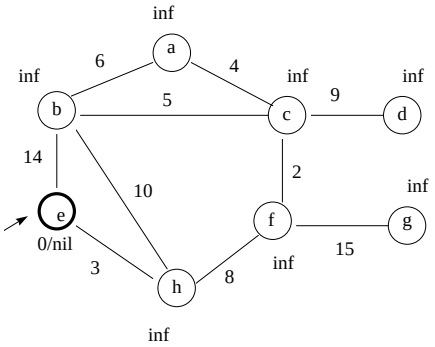
Total of ElgE over all edges for loop using the trickier O(lgE) technique
Total: $O(V) + O(E \lg E) + O(E \lg E) \rightarrow O(ElgV)$ runtime.

This algorithm is greedy because it always uses the best (smallest) edge next.

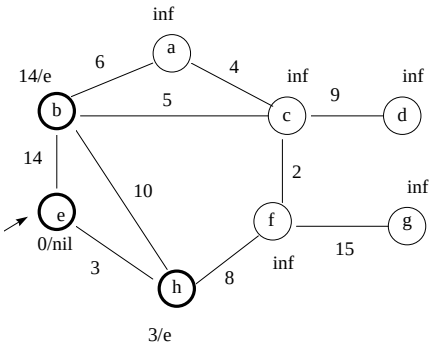
Another algorithm: Prim's MST Algorithm (also greedy)

```
MST-Prim(G,w,r)                         ; Graph G, weights w, root r
        Q ← V[G]
        for each vertex u ∈ Q do key[u] ← ∞      ; infinite "distance"
        key[r] ← 0
        P[r] ← NIL
        while Q<>NIL do
                u ← Extract-Min(Q)              ; remove closest node
                ; Update children of u so they have a parent and a min key val
                ; the key is the weight between node and parent
                for each v ∈ Adj[u] do
                        if v ∈ Q & w(u,v)<key[v] then
                                P[v] ← u
                                key[v] ← w(u,v)
```

Example: Graph given earlier.
Q={ (e,0) (a, ∞) (b, ∞) (c, ∞) (d, ∞) (f, ∞) (g, ∞) (h, ∞) }

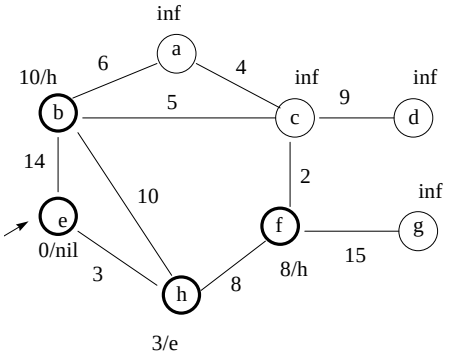


Extract min, vertex e. Update neighbor if in Q and weight < key.



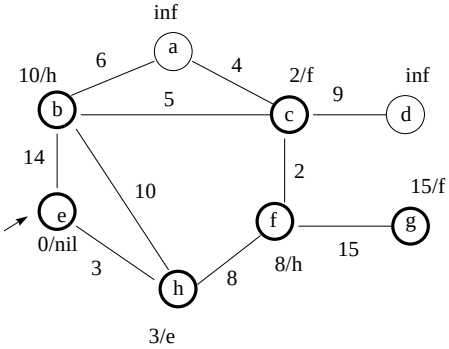
Q={ (a, ∞) (b,14) (c, ∞) (d, ∞) (f, ∞) (g, ∞) (h,3) }

Extract min, vertex h. Update neighbor if in Q and weight < key



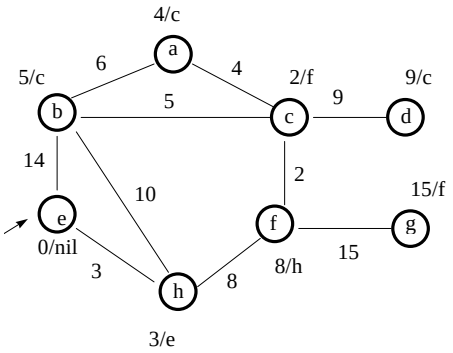
Q={ (a, ∞) (b,10) (c, ∞) (d, ∞) (f,8) (g, ∞) }

Extract min, vertex f. Update neighbor if in Q and weight < key



Q={ (a, ∞) (b,10) (c, 2) (d, ∞) (g,15) }

Extract min, vertex c. Update neighbor if in Q and weight < key



Q={ (a,4) (b,5) (d,9) (g,15) }

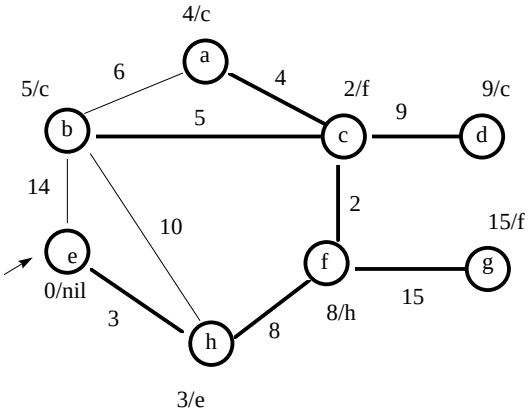
Extract min, vertex a. No keys are smaller than edges from a (4>2 on edge ac, 6>5 on edge ab) so nothing done.

Q={ (b,5) (d,9) (g,15) }

Extract min, vertex b.

Same case, no keys are smaller than edges, so nothing is done.
Same for extracting d and g, and we are done.

Get spanning tree by connecting nodes with their parents:



Note: Prim's algorithm may give a different spanning tree than Kruskal's, but both will be minimum spanning trees.

Runtime:

Initialization Loop over every vertex takes O(V)
The while Q loop goes up to V times
    Extract-Min takes O(lg V) time if we use a heap
    For loop over the Adjacency List will take O(E) time over all executions to go through each edge
        Updating the key requires some hidden operations: updating the heap so that the minimum is still correct. The implicit time is O(lg V) to update the heap.

The inner loop takes O(E lg V) for the heap update inside the O(E) loop. This is over all executions, so it is not multiplied by O(V) for the while loop (this is included in the O(E) runtime through all edges.

The Extract-Min requires O(V lg V) time. O(lg V) for the Extract-Min and O(V) for the while loop.

Total runtime is then O(V lg V) + O(E lg V) which is O(E lg V) in a connected graph (a connected graph will always have at least as many edges as vertices).

Shortest Path Algorithms

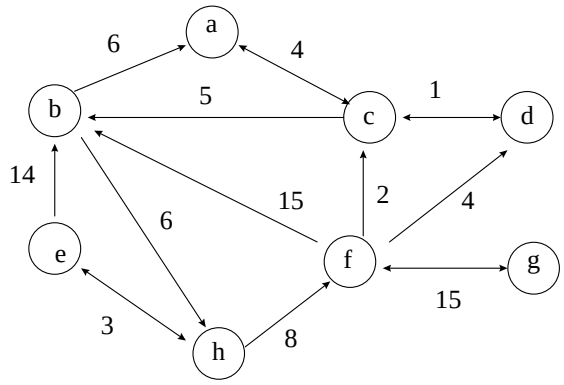
Shortest Path Algorithms

Goal: Find the shortest path between vertices in a weighted graph. We denote the shortest path between vertex u and v as δ(u,v). This is a very practical problem - the weights may represent the shortest distance, shortest time, shortest cost, etc. There are several forms of this problem:

1. Single-source shortest path. Find the shortest distance from a source vertex s to every other vertex in the graph.
2. Single-destination shortest path. Find a shortest path to a given destination vertex t from every vertex v. This is just the reverse of single-source.
3. Single-pair shortest path. Find a shortest path between a pair of vertices. No algorithm is known for this problem that runs asymptotically faster than the best single-source algorithms.
4. All-pairs shortest path. Find the shortest path between every vertex in the graph.

Note that BFS computes the shortest path on an unweighted graph.

Example: What is the shortest path from g to b?



g to f to b has cost 30, but g to f to c to b has cost 22. So the shortest path is gfcb.

Formal definition of problem:

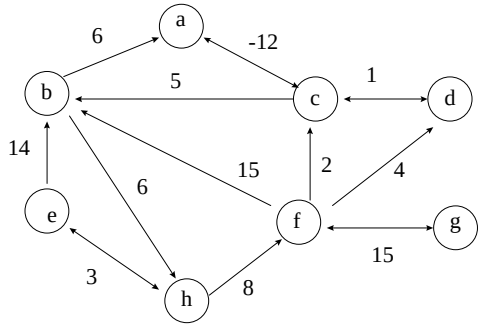
Input: Weighted directed graph G=(V,E) plus w: $E \rightarrow R$, the weight function that maps from edges to weight values (real numbers).

Output: The shortest path between any source S and all other vertices.

The weight of a path is the sum of the weights of the edges on the path.

w(u,v) is the weight of arc that connects vertex u and v; it is not necessarily the min.
w(p) is the weight of some path p, it is not necessarily the min.
δ(u,v) is the weight of the shortest path that connects vertex u and v.

- We will keep track of the parents of a vertex in P(v) then we can output either a shortest path using parents or a shortest path tree.
- A shortest path tree is a subset of the graph with the shortest path for every vertex from a source (root). This is not unique.
- We won't use negative weights – requires a different algorithm, since negative cycles can be travelled infinitely to make the weight cost lower and lower.



Ex: Can travel the a,b,c loop over and over again, each time reducing weight cost by one!

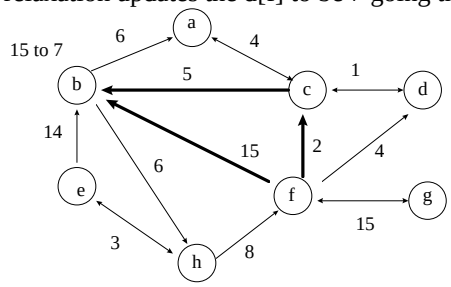
Properties of Shortest Path Problem

1. The shortest path problem has optimal substructure. That is, the subpath of a shortest path is a shortest path. If p_{ik} is the shortest path from I to K, then for any node j along this path such that $I \leq J \leq K$, p_{ij} is the shortest path from I to J. (If there was a shorter path from I to J then the IK path wasn't the shortest!)

2. $\delta(s,v) \leq \delta(s,u) + w(u,v)$

Relaxation: The process of relaxing an edge (u,v) consists of testing whether or not we can improve the shortet path to v by going through some other path through u.

Example: If we have that the distance from f to b is 15 (going through the direct edge), the process of relaxation updates the d[f] to be 7 going through c.




```
Relax(u,v,w)
        if d[v]>d[u]+w(u,v) then
                d[v] ⟵ d[u]+w(u,v)          ; decrement distance
                P[v] ⟵ u                    ; indicate parent node
```

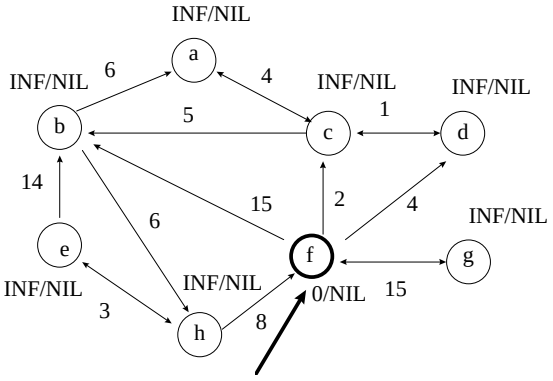
We have now covered enough to go through Dijkstra's Algorithm for finding the shortest path:

```
Dijkstra(G,w,s)              ; Graph G, weights w, source s
        for each vertex v∈G, set d[v] ← ∞ and P[v] ← NIL
        d[s] ← 0
        S ← {}
        Q ← V[G]
        while Q not empty do
                u ← Extract-Min(Q)
                S ← S ∪ {u}
                for each vertex v ∈ Adj[u] do
                        if d[v]>d[u]+w(u,v) then          ; Relax node
                                d[v] ← d[u]+w(u,v)        ; decrement distance
                                P[v] ← u                  ; indicate parent node
```

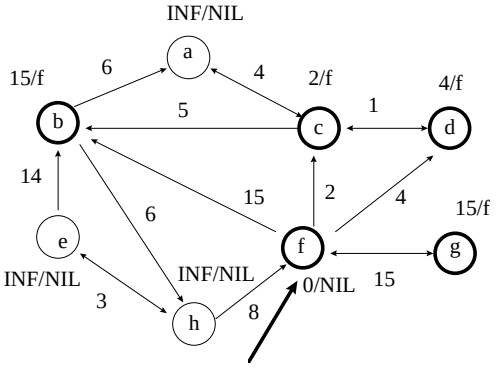
The set S has all the vertices that already have a shortest path. We keep picking u∈V-S with the minimum path estimate so far and put it into S. This makes Dijkstra's algorithm GREEDY since it picks the best estimate so far. Then we revise path estimates or the children of that vertex by relaxing all edges leaving u.

Example:

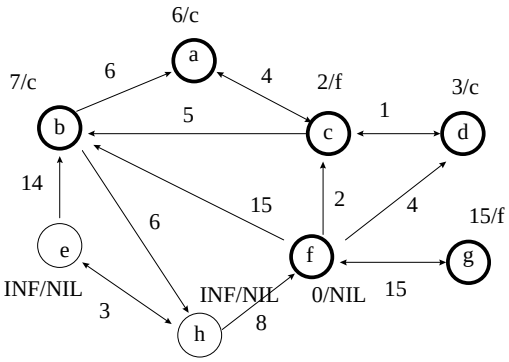
Initialize nodes to ∞, parent to nil.
S={}, Q={(a,∞) (b,∞) (c, ∞) (d, ∞) (e, ∞) (f,0) (g, ∞) (h, ∞)}



Extract Min, vertex f. S={f}, relax neighbors.
Q={(a,∞) (b,15) (c, 2) (d, 4) (e, ∞) (g, 15) (h, ∞)}



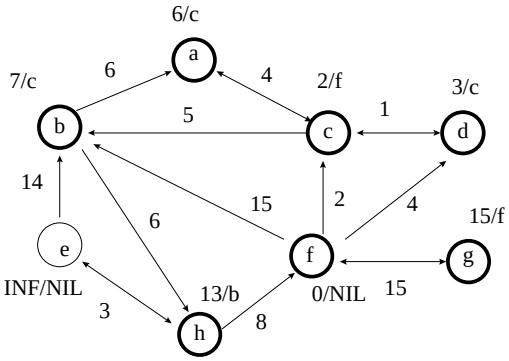
Extract Min, vertex c. S={f c}, relax neighbors.
Q={(a,6) (b,7) (d, 3) (e, ∞) (g, 15) (h, ∞)}



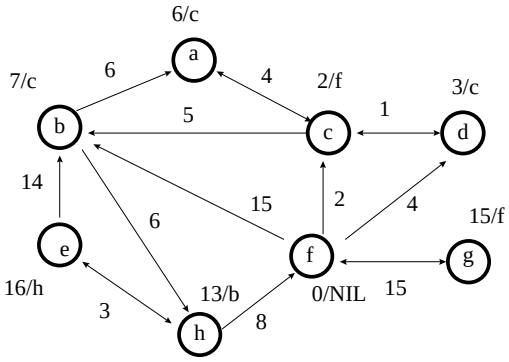
Extract Min, vertex d. S={f c d}, relax neighbors. Nothing to relax.
Q={(a,6) (b,7) (e, ∞) (g, 15) (h, ∞)}

Extract Min, vertex a. S={f c d a}, relax neighbors. Nothing to relax.
Q={ (b,7) (e, ∞) (g, 15) (h, ∞)}

Extract Min, vertex b. S={f c d a b}, relax neighbors.
Q={ (e, ∞) (g, 15) (h, 13)}



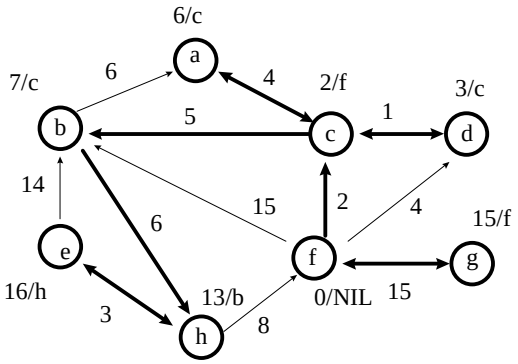
Extract Min, vertex h. S={f c d a b h}, relax neighbors.
Q={ ((e, 16) (g, 15) }



Extract Min, vertex g. S={f c d a b h g} relax neighbors. Nothing to relax.
Q={ ((e, 16) }

Extract Min, vertex e. S={f c d a b h e g}, relax neighbors. Nothing to relax.

Done! Follow parent pointers to get tree:



Comment: Very similar to Prim's MST algorithm, but the two compute different things. Not the shortest path in MST, but picks shortest edge overall to connect all edges (what case would the MST pick a different edge than Dijkstra?)

Runtime of algorithm:

If queue Q is a linear array, then Extract-Min takes time O(V).
Since every vertex is in Q exactly once, Extract-Min takes time $O(V^2)$.
The for loop looks at each edge once over all executions, so this is
executed at most O(E) times.
The time for relaxation is constant, assuming the vertices can act as
direct indices into the array (random access)
Total time: $O(V^2 + E) = O(V^2)$.

If we use a heap to do the Q, then Extract-Min takes O(lgV) time, but
the relaxation step requires updating the heap O(lgV) time, instead
of constant time.
Total time: O(VlgV + ElgV) = O((V+E)lgV).
This is O(ElgV) in a connected graph. (why?)

Note we can have $E=O(V^2)$ in a fully connected graph. This method is only preferred in a more sparse graph.

Finding All-Pairs Shortest Path:

One solution is to run Dijkstra's Algorithm for every vertex. This will require time O(V)(Time-Dijkstra). If using a linear array for Q, this is time $O(V^3)$. There are other ways to solve this problem using dynamic programming which we will examine shortly.