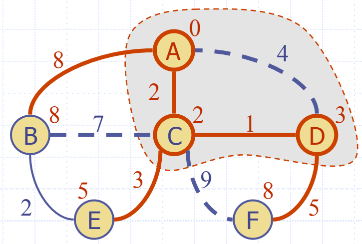
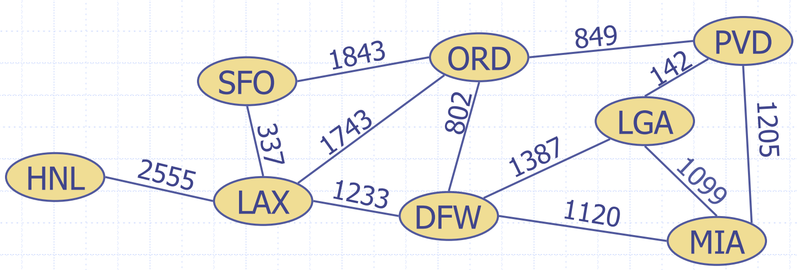
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Shortest Paths



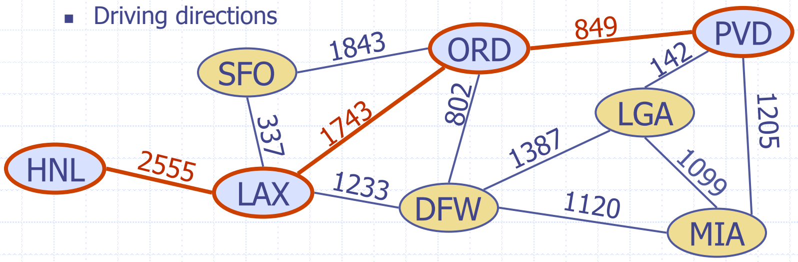
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

- Given a weighted graph and two vertices ***u*** and ***v***, we want to find a path of minimum total weight between ***u*** and ***v***.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



Shortest Path Properties

Property 1:

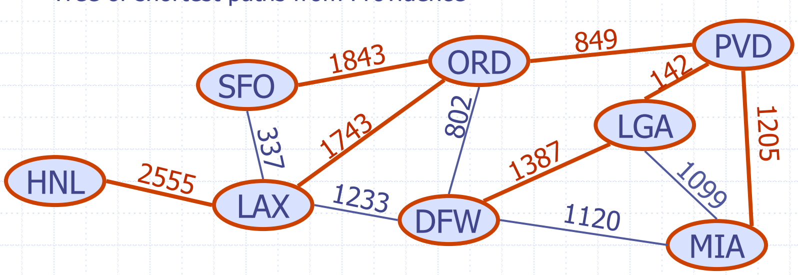
A subpath of a shortest path is itself a shortest path

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence



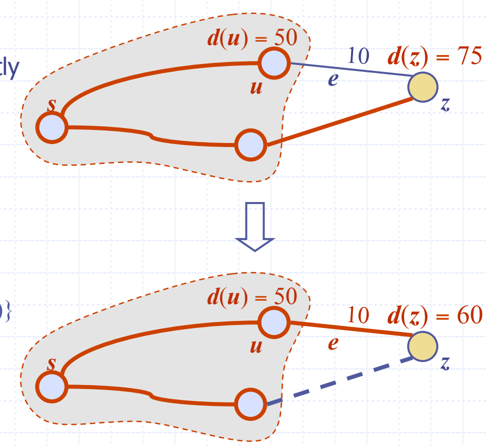
Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u

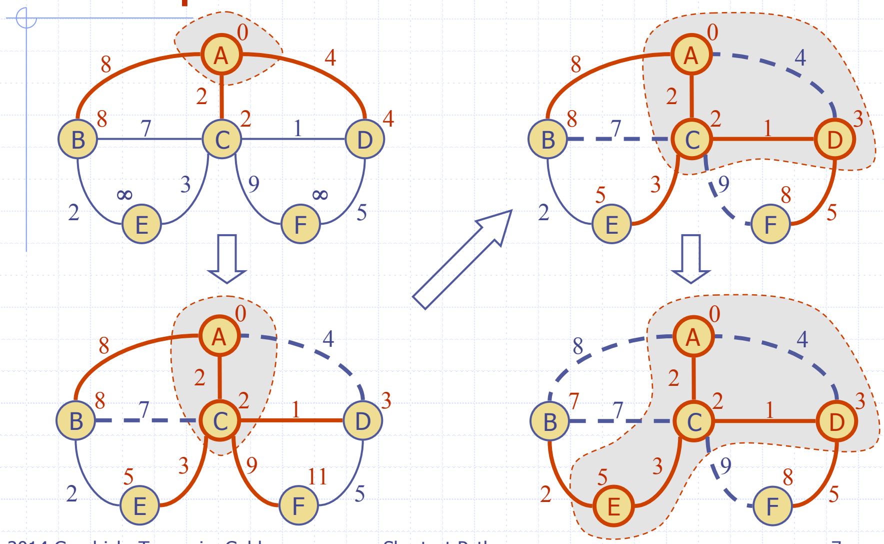
Edge Relaxation

- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

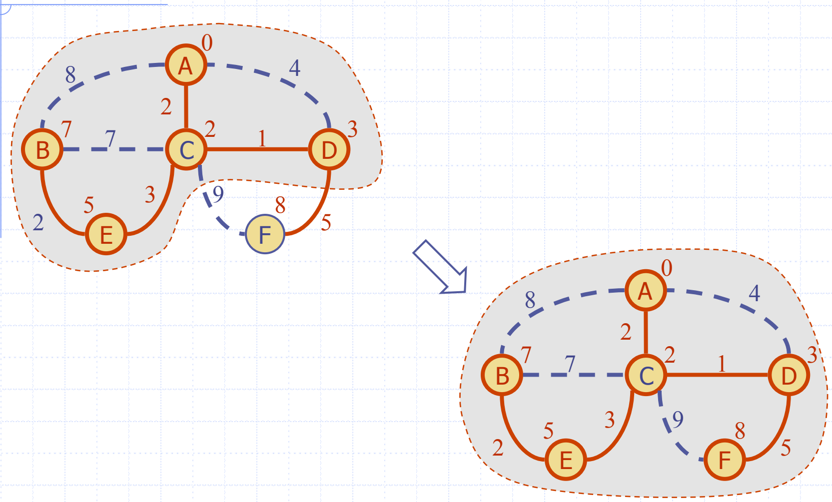
$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



Example



Example (cont.)



Dijkstra's Algorithm

Algorithm ShortestPath(G, s):
Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.
Output: The length of a shortest path from s to v for each vertex v of G.

```
Initialize D[s] = 0 and D[v] = ∞ for each vertex v ≠ s.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty do
   {pull a new vertex u into the cloud}
   u = value returned by Q.remove_min()
   for each vertex v adjacent to u such that v is in Q do
      {perform the relaxation procedure on edge (u,v)}
      if D[u] + w(u,v) < D[v] then
         D[v] = D[u] + w(u,v)
         Change to D[v] the key of vertex v in Q.
return the label D[v] of each vertex v
```


Analysis of Dijkstra's Algorithm

- Graph operations
  - We find all the incident edges once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list/map structure
  - Recall that $\Sigma_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

Java Implementation

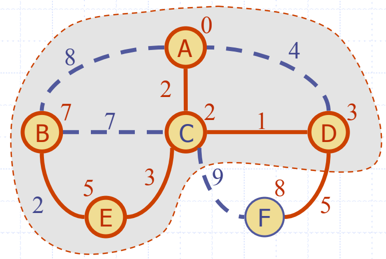
```java
/** Computes shortest-path distances from src vertex to all reachable vertices of g. */
public static <V> Map<Vertex<V>, Integer>
shortestPathLengths(Graph<V,Integer> g, Vertex<V> src) {
  // d.get(v) is upper bound on distance from src to v
  Map<Vertex<V>, Integer> d = new ProbeHashMap<>();
  // map reachable v to its d value
  Map<Vertex<V>, Integer> cloud = new ProbeHashMap<>();
  // pq will have vertices as elements, with d.get(v) as key
  AdaptablePriorityQueue<Integer, Vertex<V>> pq;
  pq = new HeapAdaptablePriorityQueue<>();
  // maps from vertex to its pq locator
  Map<Vertex<V>, Entry<Integer,Vertex<V>>> pqTokens;
  pqTokens = new ProbeHashMap<>();

  // for each vertex v of the graph, add an entry to the priority queue, with
  // the source having distance 0 and all others having infinite distance
  for (Vertex<V> v : g.vertices()) {
    if (v == src)
      d.put(v,0);
    else
      d.put(v, Integer.MAX_VALUE);
    pqTokens.put(v, pq.insert(d.get(v), v));          // save entry for future updates
  }
```


Java Implementation, 2

```java
  // now begin adding reachable vertices to the cloud
  while (!pq.isEmpty()) {
    Entry<Integer, Vertex<V>> entry = pq.removeMin();
    int key = entry.getKey();
    Vertex<V> u = entry.getValue();
    cloud.put(u, key);                          // this is actual distance to u
    pqTokens.remove(u);                         // u is no longer in pq
    for (Edge<Integer> e : g.outgoingEdges(u)) {
      Vertex<V> v = g.opposite(u,e);
      if (cloud.get(v) == null) {
        // perform relaxation step on edge (u,v)
        int wgt = e.getElement();
        if (d.get(u) + wgt < d.get(v)) {            // better path to v?
          d.put(v, d.get(u) + wgt);                 // update the distance
          pq.replaceKey(pqTokens.get(v), d.get(v)); // update the pq entry
        }
      }
    }
  }
  return cloud;          // this only includes reachable vertices
}
```

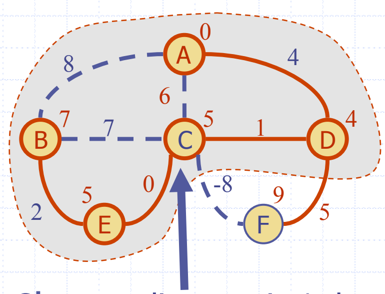
Why Dijkstra's Algorithm Works

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct
  - But the edge (D,F) was relaxed at that time!
  - Thus, so long as d(F)≥d(D), F's distance cannot be wrong. That is, there is no wrong vertex



Why It Doesn't Work for Negative-Weight Edges

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with d(C)=5!

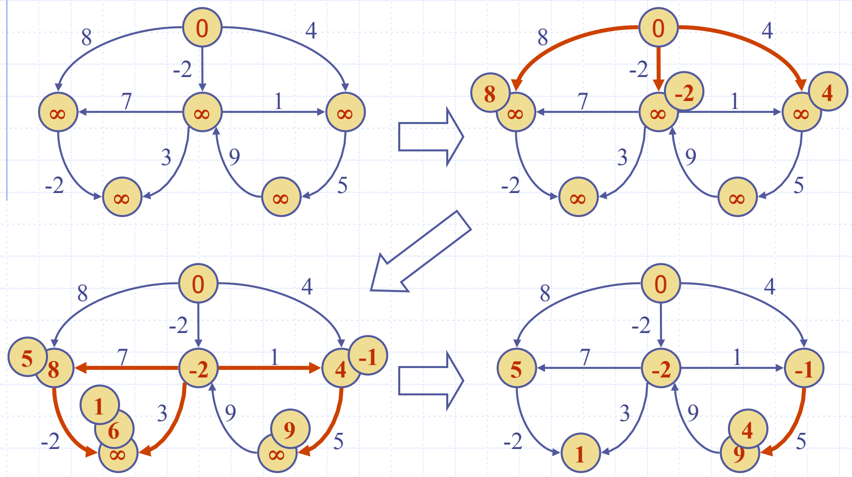
Bellman-Ford Algorithm (not in book)

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: O(nm).
- Can be extended to detect a negative-weight cycle if it exists
  - How?

```
Algorithm BellmanFord(G, s)
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  for i ← 1 to n − 1 do
    for each e ∈ G.edges()
      { relax edge e }
      u ← G.origin(e)
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
```


Bellman-Ford Example

Nodes are labeled with their d(v) values



DAG-based Algorithm (not in book)

- Works even with negative-weight edges
- Uses topological order
- Doesn't use any fancy data structures
- Is much faster than Dijkstra's algorithm
- Running time: O(n+m).

```
Algorithm DagDistances(G, s)
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  { Perform a topological sort of the vertices }
  for u ← 1 to n do    {in topological order}
    for each e ∈ G.outEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
```


DAG Example

Nodes are labeled with their d(v) values