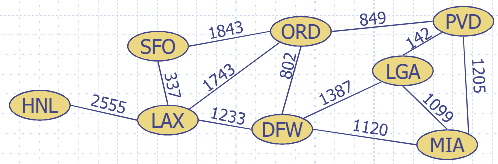
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Shortest Paths

Lightning strike, 2009. U.S. government image. NOAA.

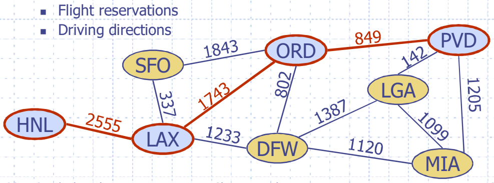
Weighted Graphs

- 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:

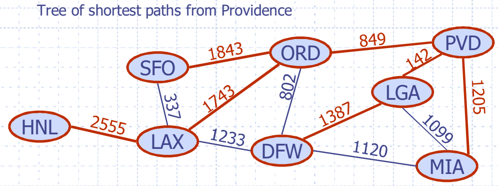
A subpath of a shortest path is itself a shortest path

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



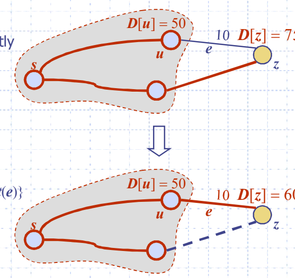
Dijkstra's Algorithm

- 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:

$$D[z] \leftarrow \min\{D[z], D[u] + weight(e)\}$$



Dijkstra’s Algorithm: Details

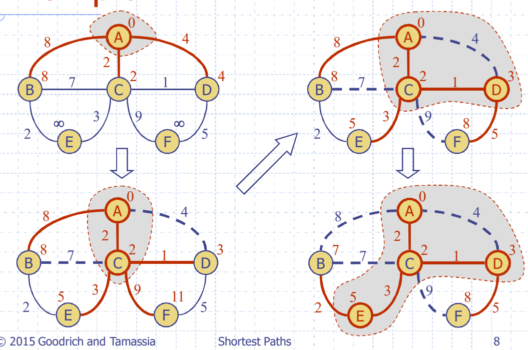
Algorithm DijkstraShortestPaths(G, v):

    Input: A simple undirected weighted graph G with nonnegative edge weights, and a distinguished vertex v of G

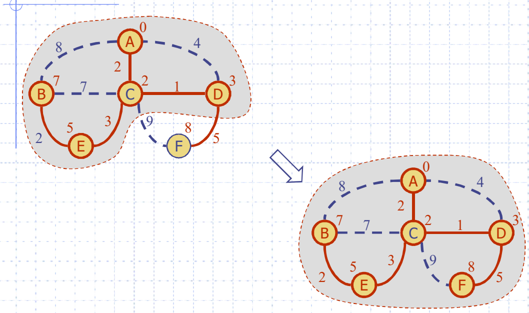
    Output: A label, $D[u]$, for each vertex u of G, such that $D[u]$ is the distance from v to u in G

    $D[v] \leftarrow 0$
    for each vertex $u \neq v$ of G **do**
        $D[u] \leftarrow +\infty$
    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
        $u \leftarrow Q$.removeMin()
        for each vertex z adjacent to u such that z is in Q **do**
            // perform the ***relaxation*** procedure on edge (u, z)
            if $D[u] + w((u, z)) < D[z]$ **then**
                $D[z] \leftarrow D[u] + w((u, z))$
                Change the key for vertex z in Q to $D[z]$
    return the label $D[u]$ of each vertex u

Example



Example (cont.)

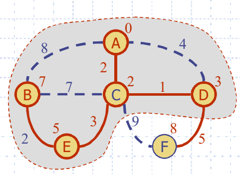


Analysis of Dijkstra’s Algorithm

- 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 $\sum_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

Why Dijkstra’s Algorithm Works

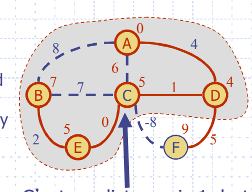
- Dijkstra’s algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn’t find all shortest distances. Let w be the first wrong vertex the algorithm processed.
  - When the previous node, u, on the true shortest path was considered, its distance was correct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



(u,w) = (D,F) in this example

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

- 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?

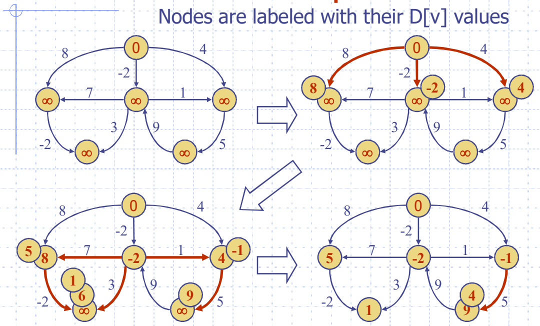
Bellman-Ford Algorithm: Details

Algorithm BellmanFordShortestPaths($\vec{G}, v$):
Input: A weighted directed graph $\vec{G}$ with n vertices, and a vertex v of $\vec{G}$
Output: A label $D[u]$, for each vertex u of $\vec{G}$, such that $D[u]$ is the distance from v to u in $\vec{G}$, or an indication that $\vec{G}$ has a negative-weight cycle

$D[v] \leftarrow 0$
for each vertex $u \neq v$ of $\vec{G}$ **do**
  $D[u] \leftarrow +\infty$
for $i \leftarrow 1$ to $n-1$ **do**
  for each (directed) edge (u, z) outgoing from u **do**
    // Perform the ***relaxation*** operation on (u, z)
    if $D[u] + w((u, z)) < D[z]$ **then**
      $D[z] \leftarrow D[u] + w((u, z))$
if there are no edges left with potential relaxation operations **then**
  return the label $D[u]$ of each vertex u
else
  return "$\vec{G}$ contains a negative-weight cycle"

Bellman-Ford Example

Nodes are labeled with their D[v] values



DAG-based Algorithm

- We can produce a specialized shortest-path algorithm for directed acyclic graphs (DAGs)
- 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).

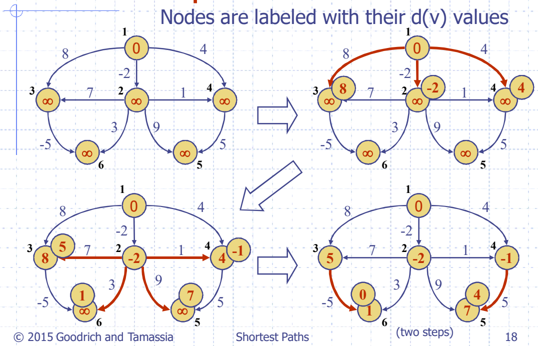
DAG-based Algorithm: Details

Algorithm DAGShortestPaths($\vec{G}, s$):
Input: A weighted directed acyclic graph (DAG) $\vec{G}$ with n vertices and m edges, and a distinguished vertex s in $\vec{G}$
Output: A label $D[u]$, for each vertex u of $\vec{G}$, such that $D[u]$ is the distance from v to u in $\vec{G}$

Compute a topological ordering $(v_1, v_2, \ldots, v_n)$ for $\vec{G}$
$D[s] \leftarrow 0$
for each vertex $u \neq s$ of $\vec{G}$ **do**
  $D[u] \leftarrow +\infty$
for $i \leftarrow 1$ to $n-1$ **do**
  // Relax each outgoing edge from v_i
  for each edge (v_i, u) outgoing from v_i **do**
    if $D[v_i] + w((v_i, u)) < D[u]$ **then**
      $D[u] \leftarrow D[v_i] + w((v_i, u))$
Output the distance labels D as the distances from s.

DAG Example

Nodes are labeled with their d(v) values



All-Pairs Shortest Paths

- Find the distance between every pair of vertices in a weighted directed graph G.
- We can make n calls to Dijkstra's algorithm (if no negative edges), which takes O(nmlog n) time.
- Likewise, n calls to Bellman-Ford would take O(n^2m) time.
- We can achieve O(n^3) time using dynamic programming (similar to the Floyd-Warshall algorithm).

```
Algorithm AllPair(G) {assumes vertices 1,…,n}
 for all  vertex pairs (i,j)
   if  i = j
      D_0[i,i] ← 0
   else if (i,j) is an edge in G
      D_0[i,j] ← weight of edge (i,j)
   else
      D_0[i,j] ← + ∞
 for k ← 1 to n do
   for i ← 1 to n do
     for j ← 1 to n do
       D_k[i,j] ← min{D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j]}
 return D_n
```