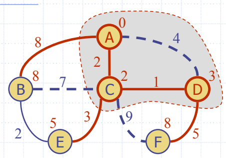
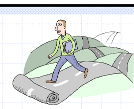
Shortest Paths



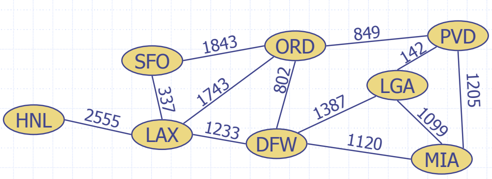
Outline and Reading

- Weighted graphs (§7.1)
  - Shortest path problem
  - Shortest path properties
- Dijkstra's algorithm (§7.1.1)
  - Algorithm
  - Edge relaxation
- The Bellman-Ford algorithm (§7.1.2)
- Shortest paths in dags (§7.1.3)
- All-pairs shortest paths (§7.2.1)

Weighted Graphs

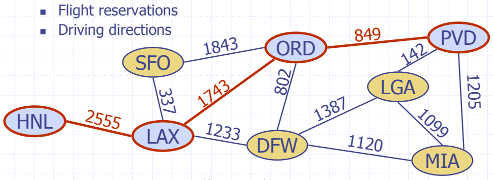


- In a weighted graph, each edge has a weight (an associated numerical value)
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports

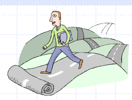


Shortest Path Problem

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



Shortest Path Properties



Property 1:

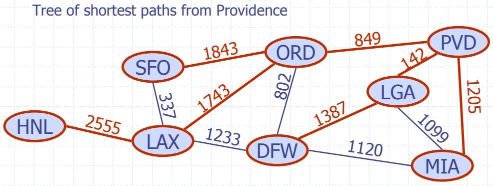
A subpath of a shortest path is itself a shortest path

Property 2:

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

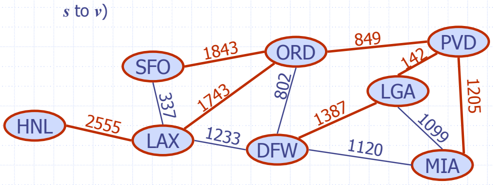
Example:

Tree of shortest paths from Providence



Single-Source Shortest Paths Problem

- Given a weighted graph and one source vertex s, find the shortest path tree T.
- T is a tree rooted at s representing shortest path from s to every other vertex v in the graph.
  - (The simple path from s to v in tree T is a shortest path from s to v)



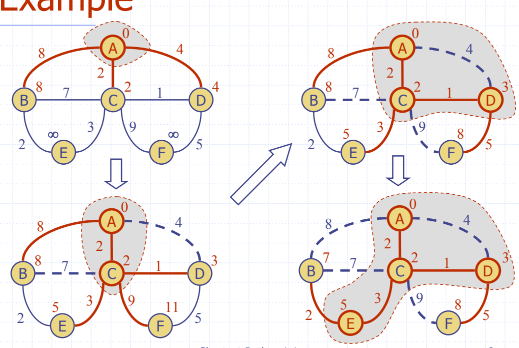
Dijkstra's Algorithm

- Solves single-source shortest path problem
- Also computes distances from source vertex s to other vertices v
- Is a greedy algorithm
- Assumptions:
  - the graph is connected
  - the edge weights are nonnegative
  - (in example, the edges are undirected)

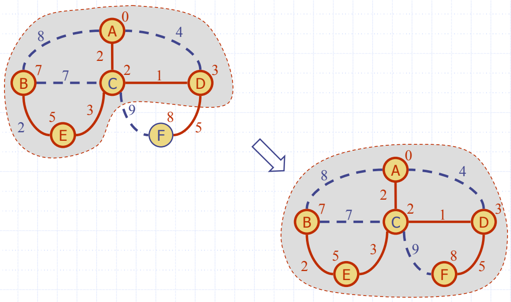
Dijkstra's Algorithm

- We grow a "**cloud**" of vertices, beginning with s and eventually covering all the vertices
- "**cloud**" of vertices contains shortest path tree
- Store $d(v)$ at each vertex v; $d(v)$ represents the distance of v from s in the "**cloud** + adjacent vertices" subgraph
- Also track edge used to get to v
- At each step
  - Add outside vertex u with the smallest distance $d(u)$ into cloud
  - Update distance labels (= several edge relaxation steps)

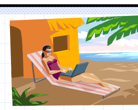
Example



Example (cont.)

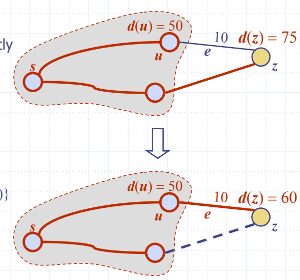


Edge Relaxation



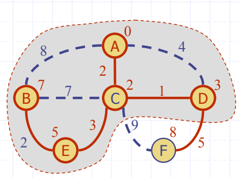
- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

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



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.



Dijkstra's Algorithm

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - *insert*(*k*,*e*) returns a locator
  - *replaceKey*(*l*,*k*) changes the key of an item
- We store three labels with each vertex:
  - Distance (d(v) label)
  - locator in priority queue
  - Edge used to get there (tree edge)

```
Algorithm Dijkstra (G, s)
  Q ← new heap-based priority queue
  for all v ∈ G.vertices()
    if v = s then  setDistance(v, 0)
    else           setDistance(v, ∞)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
    setTreeEdge(v, ∅)
  while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        Q.replaceKey(getLocator(z),r)
        setTreeEdge(z,e)
```

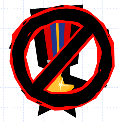

Analysis 1

- Graph operations using adjacency list structure: $O(m)$ time
  - incidentEdges iterates through incident edges once for each vertex:
- Label operations: $O(m)$ time
  - We set/get the labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations (heap-based): $O(n \log n + m \log n)$
  - Insert and remove happens once for each vertex; at cost $O(\log n)$ time each.
  - key of any vertex w modified up to $\deg(w)$ times, at cost $O(\log n)$ time
- Dijkstra's algorithm runs in $O(m \log n)$ time provided
  - the graph is connected
  - graph represented by the adjacency list structure
  - we use heap-based PQ

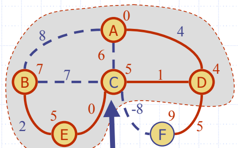
Analysis 2

- Graph operations using adjacency list structure: $O(m)$ time
  - incidentEdges iterates through incident edges once for each vertex:
- Label operations: $O(m)$ time
  - We set/get the labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations (unsorted sequence): $O(n^2 + m)$
  - Insert and remove happens once for each vertex; at cost $O(n)$ time each
  - key of any vertex w modified up to $\deg(w)$ times, at cost $O(1)$ each time
- Dijkstra's algorithm runs in $O(n^2)$ time provided
  - the graph is connected
  - graph represented by the adjacency list structure
  - we use unsorted-sequence based PQ

Why It Doesn't Work for Negative-Weight Edges

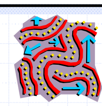


- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



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

Bellman-Ford Algorithm

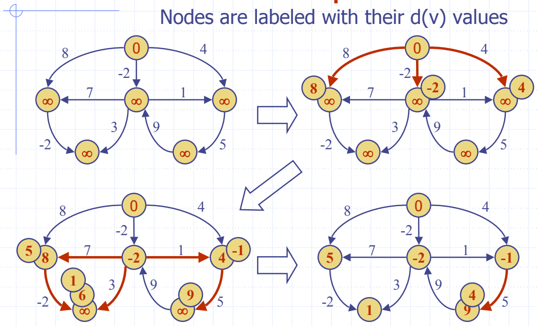


- Works even with negative-weight edges (on directed graphs)
- Iteration i finds all shortest paths that use i edges.
- Running time: O(nm).
- Can be extended to detect a negative-weight cycle if it exists
  - How?

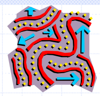
```
Algorithm BellmanFord(G, s)
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  for i ← 1 to n-1 do
    for each e ∈ G.edges()
      { relax edge e }
      u ← G.origin(e)
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)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


Bellman-Ford Example

Nodes are labeled with their d(v) values



DAG-based Algorithm

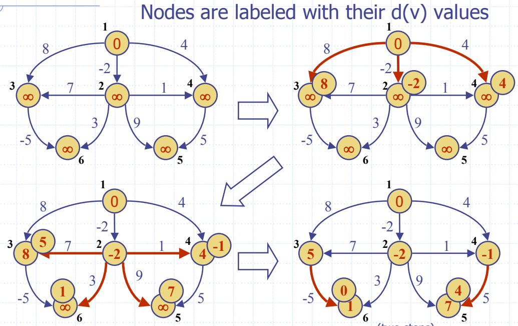


- Only for DAGs
- Works even with negative-weight edges
- Uses topological order
- Doesn't use any fancy data structures
- Is much faster than Dijkstra's algorithm
- Running time: O(n+m).

```
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



(two steps)

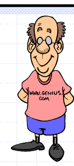
All-Pairs Shortest Paths

- Find the distance between every pair of vertices in a weighted directed graph G.
- number vertices in G: 1,2,…, n
- Store as a matrix D, so D[i,j] represents cost of shortest path from i to j.
- Distance may be infinite, meaning no path.
- Possible solutions:
  - Use Dijkstra's algorithm n times, one for each vertex
    - Only works if no negative edges
    - takes O(nmlog n) time.
  - Use Bellman-Ford n times, one for each vertex
    - takes $O(n^2m)$ time.
  - $O(n^3)$ time with Floyd-Warshall

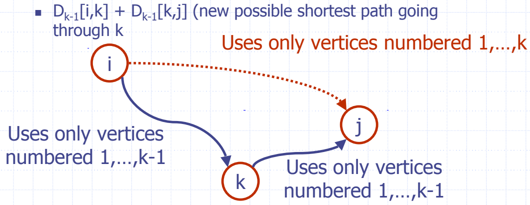
Floyd-Warshall's Algorithm

- Extension of reachability algorithm
- Based on similar recurrence:
  - Let $D_k[i,j]$ denote cost of shortest path from i to j whose intermediate vertices are a subset of {1,2,…,k}.
  - Then $D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$.
- What is $D_0[i,j]$? What is $D_n[i,j]$?

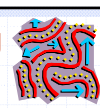
Floyd-Warshall All-Pairs shortest paths



- Computing D_k from D_{k-1} :
- For each pair of vertices (i,j) in D_{k-1} set $D_k[i,j]$ to minimum of
  - $D_{k-1}[i,j]$ (previous shortest path)
  - $D_{k-1}[i,k] + D_{k-1}[k,j]$ (new possible shortest path going through k



All-Pairs Shortest Paths using Floyd-Warshall



- Non-recursive dynamic programming version of Floyd-Warshall
- $O(n^3)$ time

```
Algorithm AllPair(G) {assumes vertices 1,…,n}
 for all vertex pairs (i,j)
   if i = j
      D0[i,i] ← 0
   else if (i,j) is an edge in G
      D0[i,j] ← weight of edge (i,j)
   else
      D0[i,j] ← + ∞
 for k ← 1 to n do
   for i ← 1 to n do
     for j ← 1 to n do
       Dk[i,j] ← min{Dk-1[i,j], Dk-1[i,k]+Dk-1[k,j]}
 return Dn
```