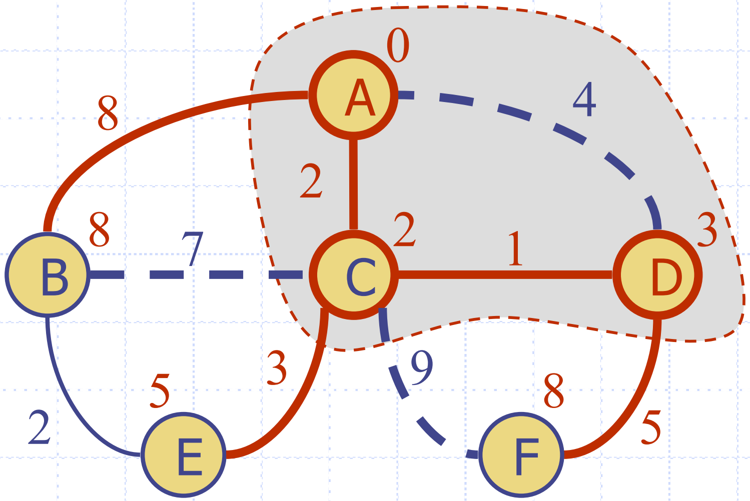
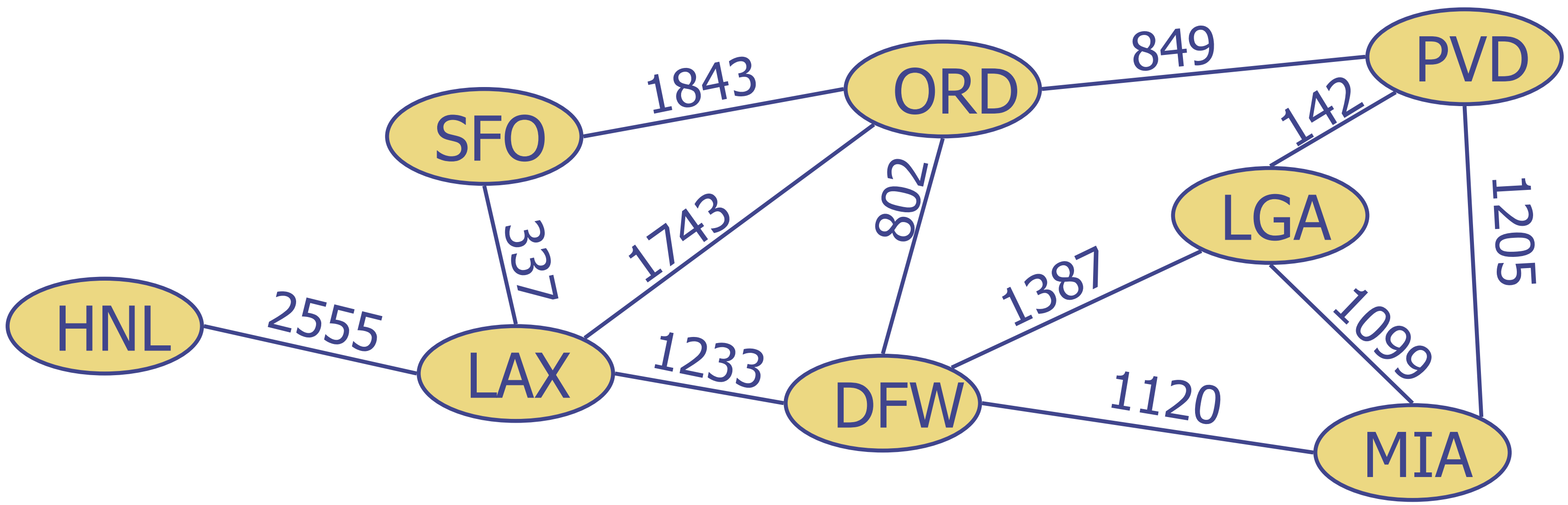
Shortest Paths



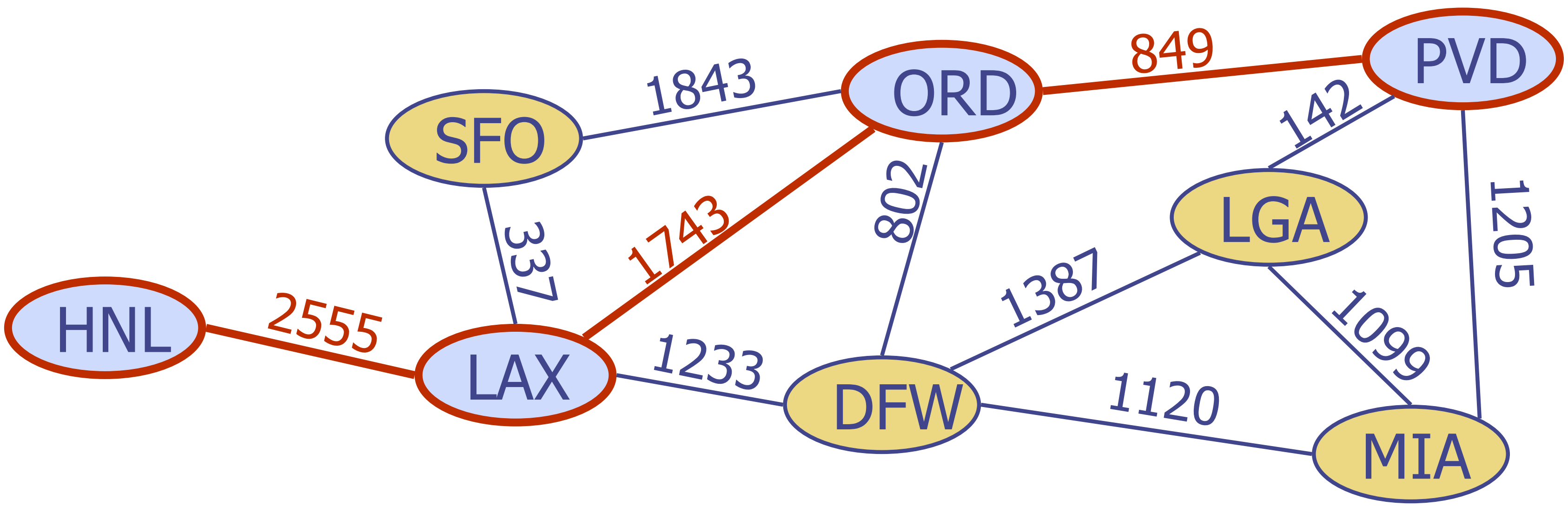
Weighted Graphs

- 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:

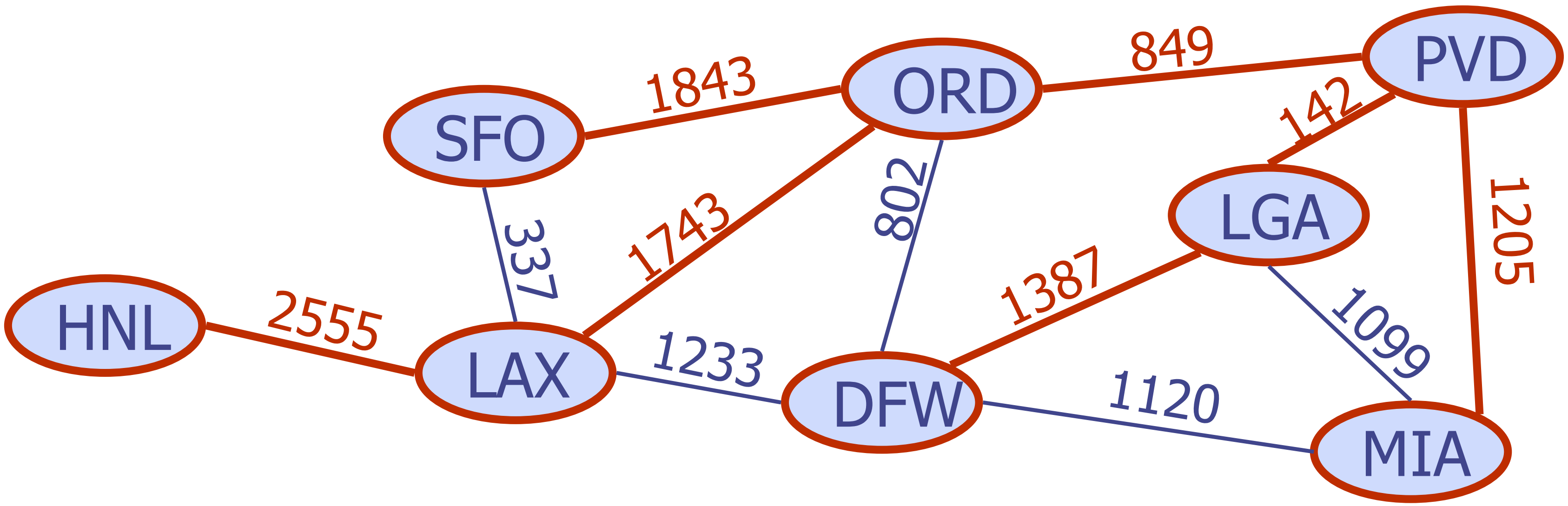
A subpath of a shortest path is itself a shortest path

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 (PVD)



Our goal and Initial Ideas

- Goal
  - Given a source vertex s, I want to compute the shortest paths to all other vertices

- Initial Ideas
  - Compute all the paths from the source ***s*** to other vertices
  - Take the minimums
  - How much complexity?
    - Exponential (not a polynomial time algorithm)
  - Why is this algorithm stupid?
    - Ignore the wisdom from computing the minimum path for computing other minimum paths

Dijkstra's Algorithm (1)

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v

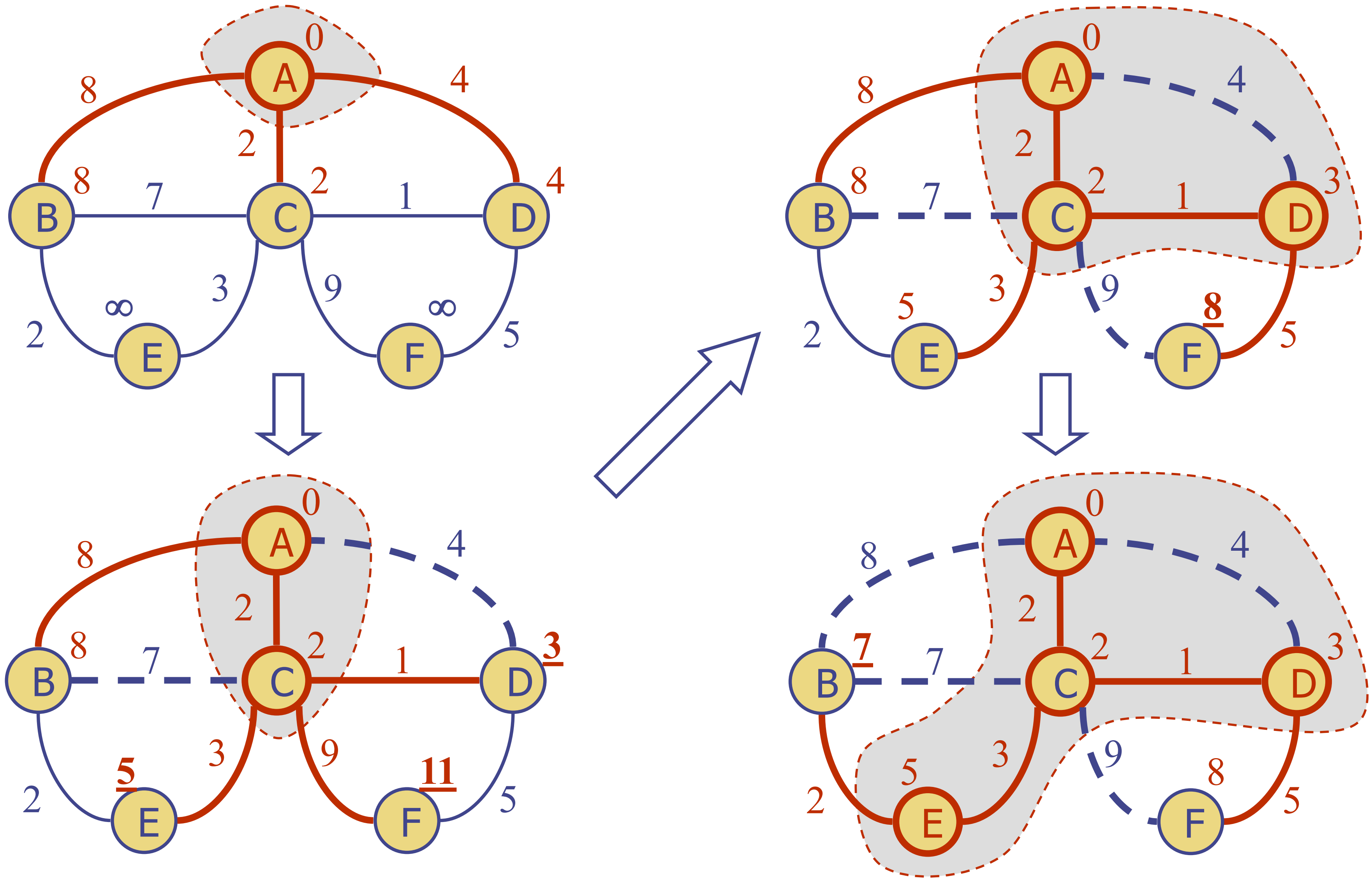
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s

- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative

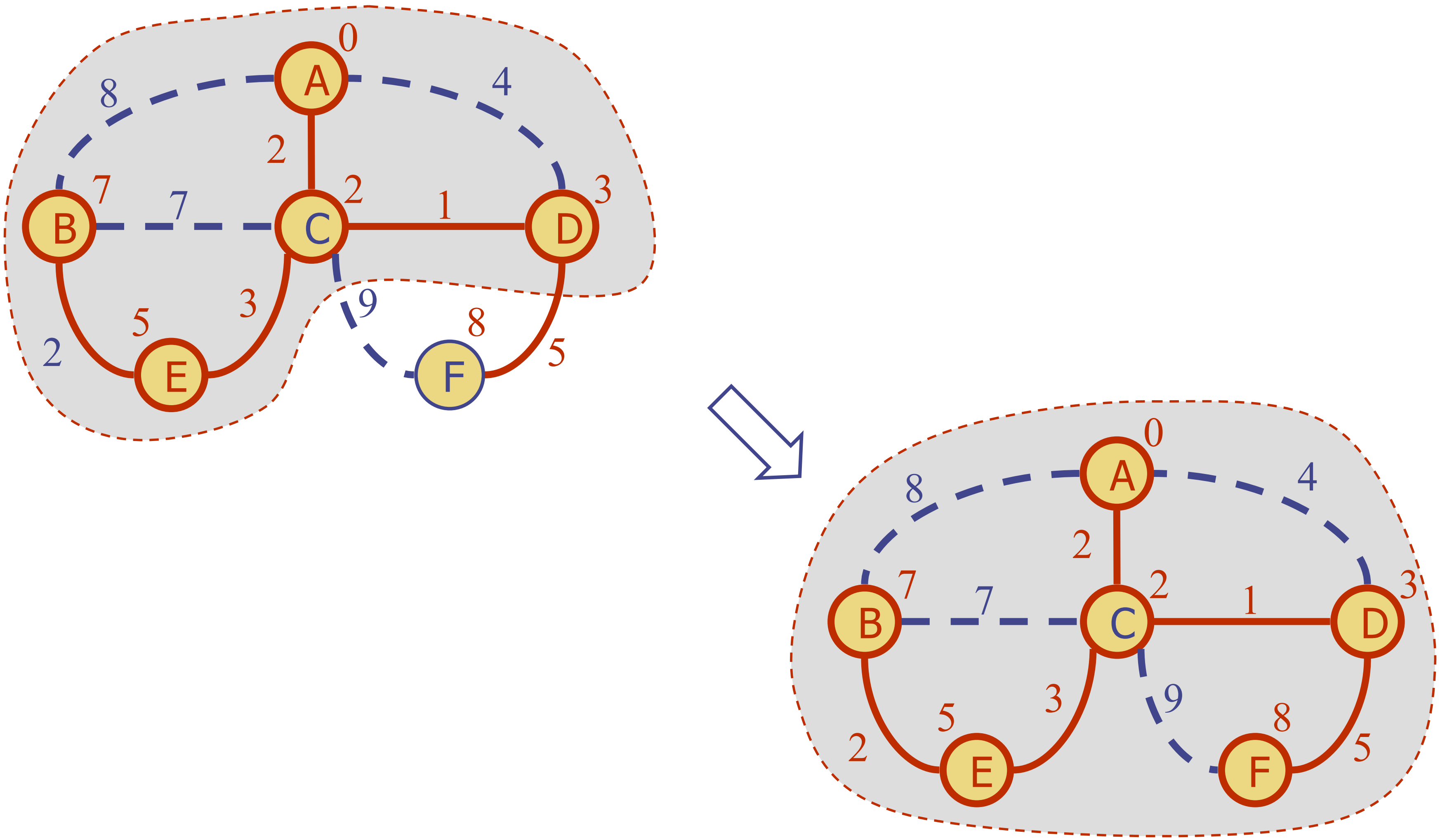
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
  - Remember the "wisdom"

- Example
  - What is your distance to "Obama" in facebook? 50
  - Suppose that MoonJaein becomes your friend
  - What is your distance to "Obama" then?
    - Probably much shorter than 50. Maybe 2? ☺

Example first



Example (cont.)



Dijkstra's Algorithm (2)

- 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
  - Greedy method: we solve the problem at hand by repeatedly selecting the best choice from among those available in each iteration

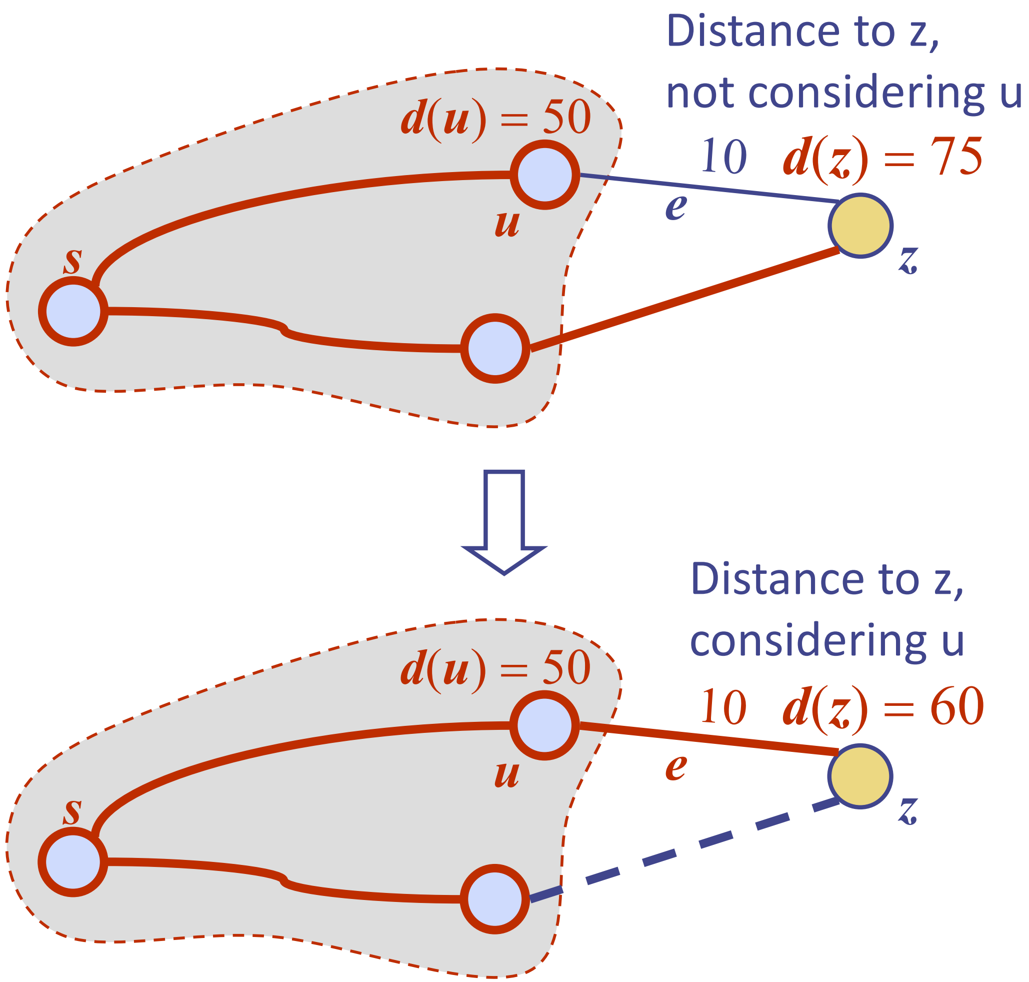
> Raise your quality standards as high as you can live with, avoid wasting your time on routine problems, and always try to work as closely as possible at the boundary of your abilities. Do this, because it is the only way of discovering how that boundary should be moved forward.
>
> Edsger Dijkstra

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:

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



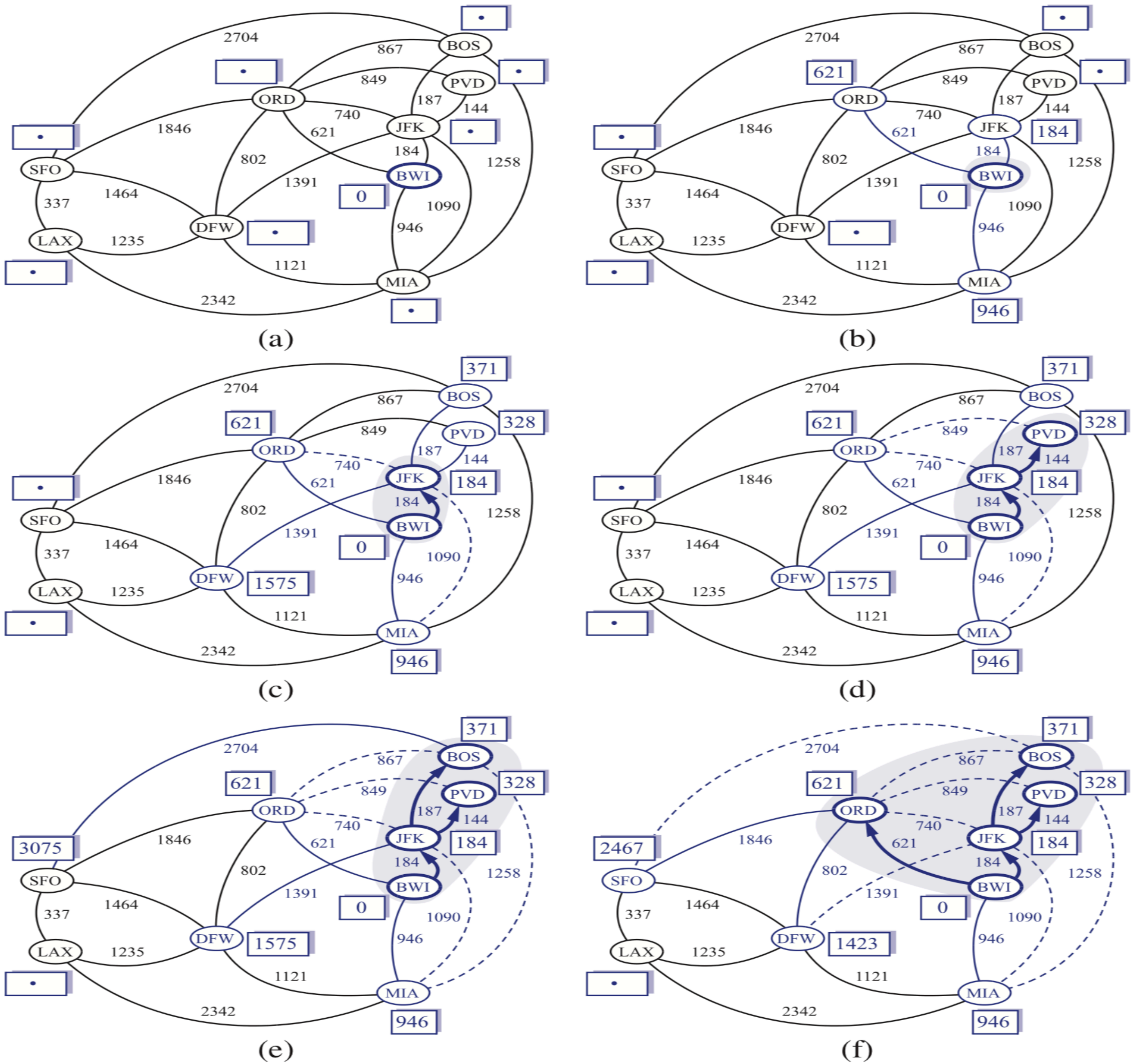
Recall: Priority Queue ADT

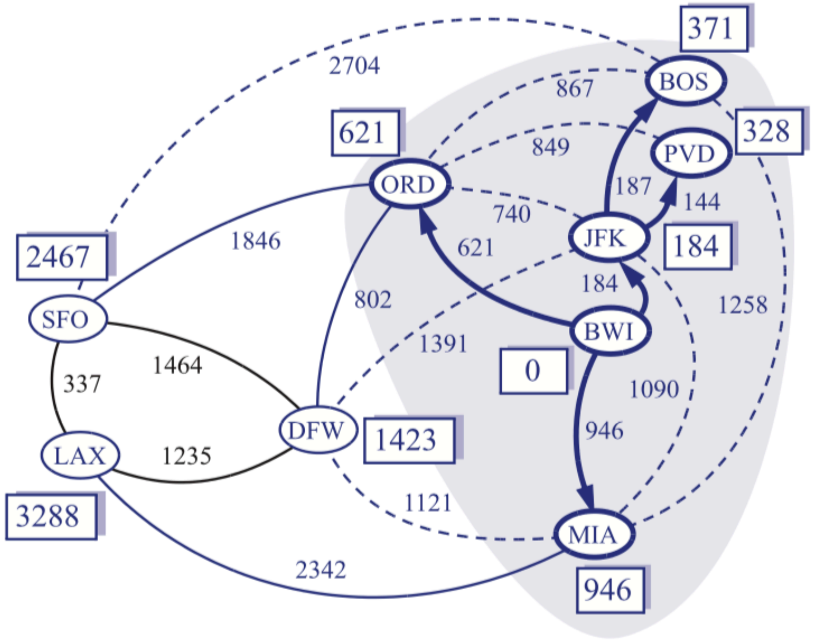
- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - insert(e) inserts an entry e
  - removeMin()
    removes the entry with smallest key
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Dijkstra's Algorithm

- A heap-based adaptable priority queue with location-aware entries stores the vertices outside the cloud
  - Key: distance
  - Value: vertex
  - Recall that method *replaceKey*(*l*,*k*) changes the key of entry *l* with *k*
- We store two labels with each vertex:
  - Distance
  - Entry in priority queue
- We take out the vertex with the minimum distance so far

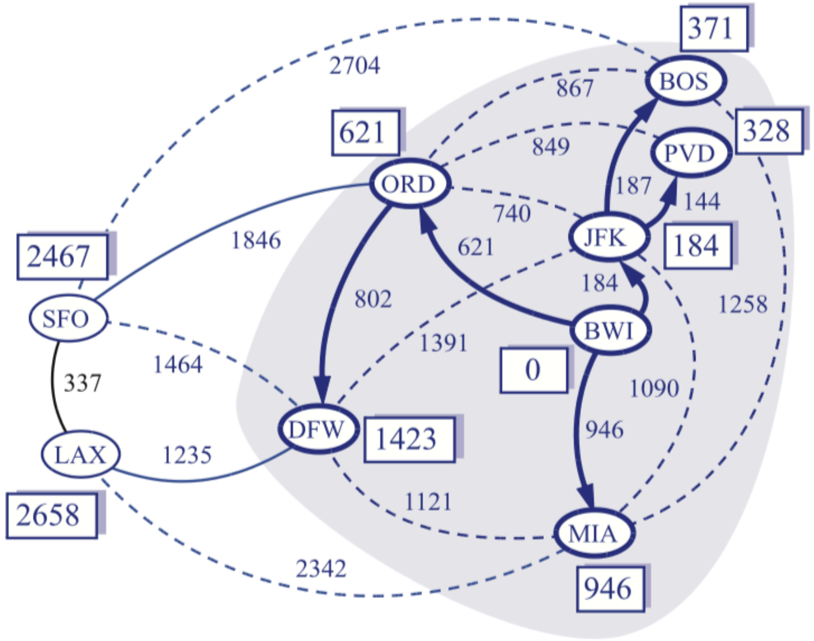
```
Algorithm DijkstraDistances(G, s)
  Q ← new heap-based priority queue
  for all v ∈ G.vertices()
    if v = s
      v.setDistance(0)
    else
      v.setDistance(∞)
    l ← Q.insert(v.getDistance(), v)
    v.setEntry(l)
  while ¬Q.empty()
    l ← Q.removeMin()
    u ← l.getValue() // take out the closest node
    for all e ∈ u.incidentEdges() { relax e }
      z ← e.opposite(u)
      r ← u.getDistance() + e.weight()
      if r < z.getDistance()
        z.setDistance(r)
        Q.replaceKey(z.getEntry(), r)
```


BOS
PVD
ORD
JFK
BWI
SFO
LAX
DFW
MIA
2704
867
849
187
144
740
621
1846
802
184
1258
1391
1464
337
1090
946
1235
1121
2342
0
621
184
946
371
328
1575
3075
2467
1423
(a)
(b)
(c)
(d)
(e)
(f)



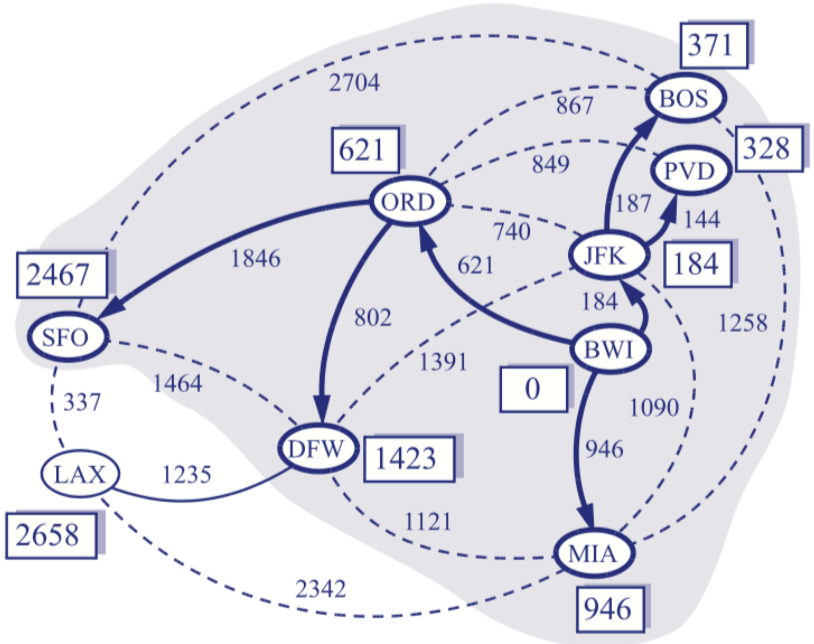
371
BOS
2704
867
328
621
849
PVD
ORD
187
144
740
JFK
184
2467
1846
621
184
802
1258
SFO
BWI
1464
1391
0
337
1090
DFW
1423
946
LAX
1235
1121
3288
MIA
2342
946

(g)



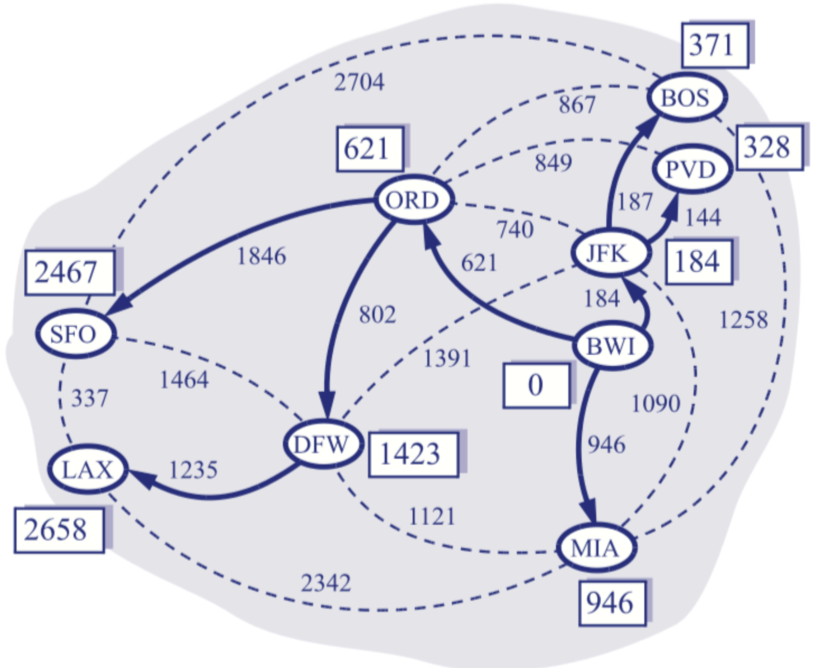
371
BOS
2704
867
328
621
849
PVD
ORD
187
144
740
JFK
184
2467
1846
621
184
802
1258
SFO
BWI
1464
1391
0
337
1090
DFW
1423
946
LAX
1235
1121
2658
MIA
2342
946

(h)



371
BOS
2704
867
328
621
849
PVD
ORD
187
144
740
JFK
184
2467
1846
621
184
802
1258
SFO
BWI
1464
1391
0
337
1090
DFW
1423
946
LAX
1235
1121
2658
MIA
2342
946

(i)



371
BOS
2704
867
328
621
849
PVD
ORD
187
144
740
JFK
184
2467
1846
621
184
802
1258
SFO
BWI
1464
1391
0
337
1090
DFW
1423
946
LAX
1235
1121
2658
MIA
2342
946

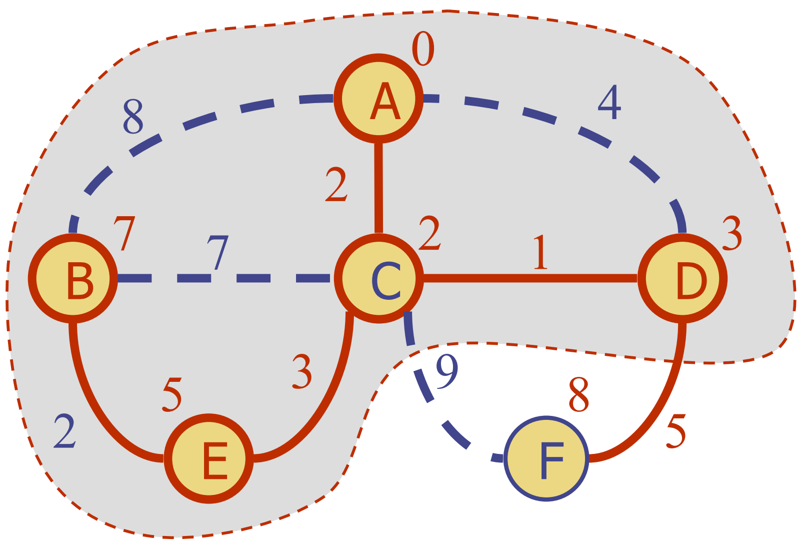
(j)

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

- (Question) Why not working for non-negative weight?



Analysis of Dijkstra's Algorithm

- Graph operations
  - incidentEdges is called once for each $\boldsymbol{v}$
- 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 $\boldsymbol{v}$ is inserted once into and removed once from the PQ, where each insertion or removal takes $O(\log n)$ time → total $nO(\log n)$
  - The key of a vertex in the PQ is modified at most $\deg(v)$ 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 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

```
Algorithm DijkstraDistances(G, s)
  Q ← new heap-based priority queue
  for all v ∈ G.vertices()
    if v = s
      v.setDistance(0)
    else
      v.setDistance(∞)
    l ← Q.insert(v.getDistance(), v)
    v.setEntry(l)
  while ¬Q.empty()
    l ← Q.removeMin()
    u ← l.getValue() // take out the closest node
    for all e ∈ u.incidentEdges() { relax e }
      z ← e.opposite(u)
      r ← u.getDistance() + e.weight()
      if r < z.getDistance()
        z.setDistance(r)
        Q.replaceKey(z.getEntry(), r)
```

Questions?