



CSE408

Single Source Shortest path

Lecture #28

Outline and Reading

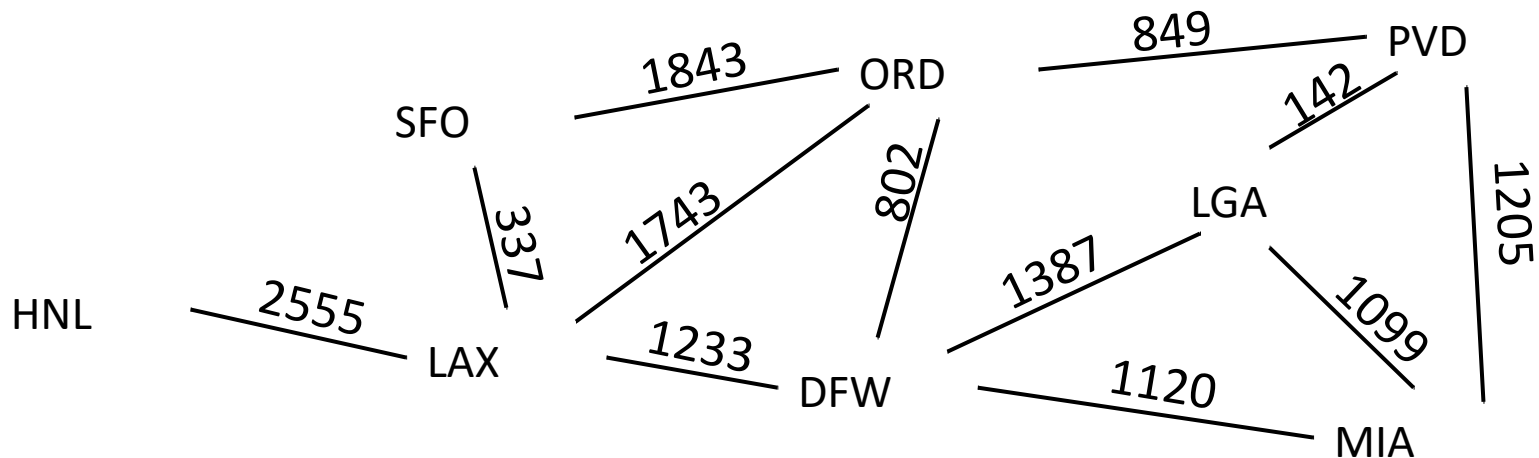


- Weighted graphs
 - Shortest path problem
 - Shortest path properties
- Dijkstra's algorithm
 - Algorithm
 - Edge relaxation
- The Bellman-Ford algorithm
- All-pairs shortest paths

Weighted Graphs



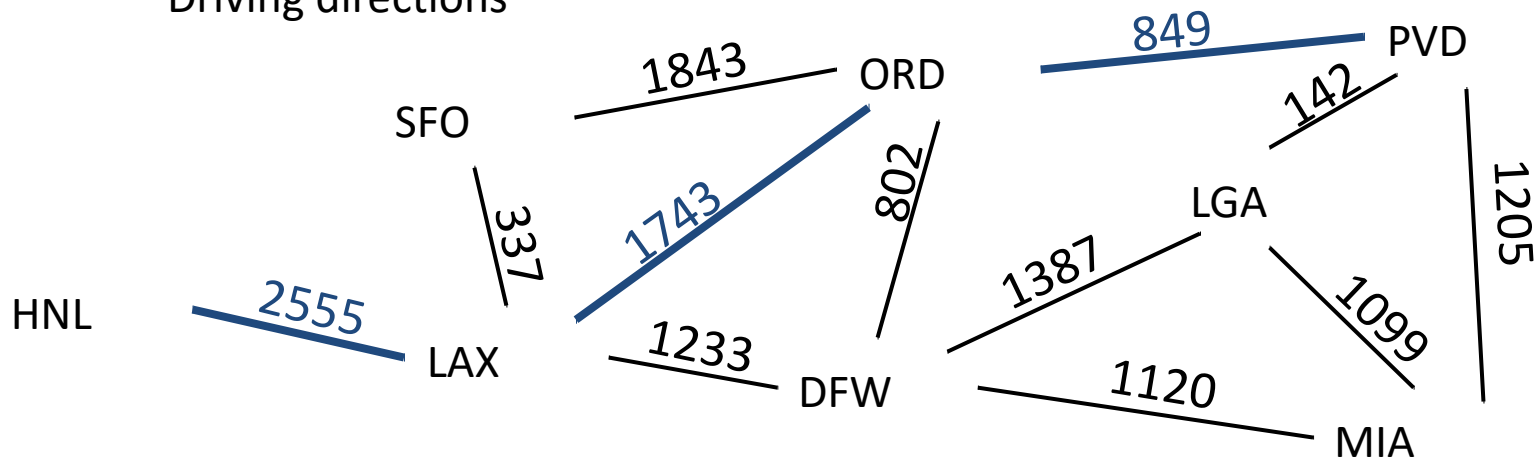
- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
 - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Path Problem



- 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:

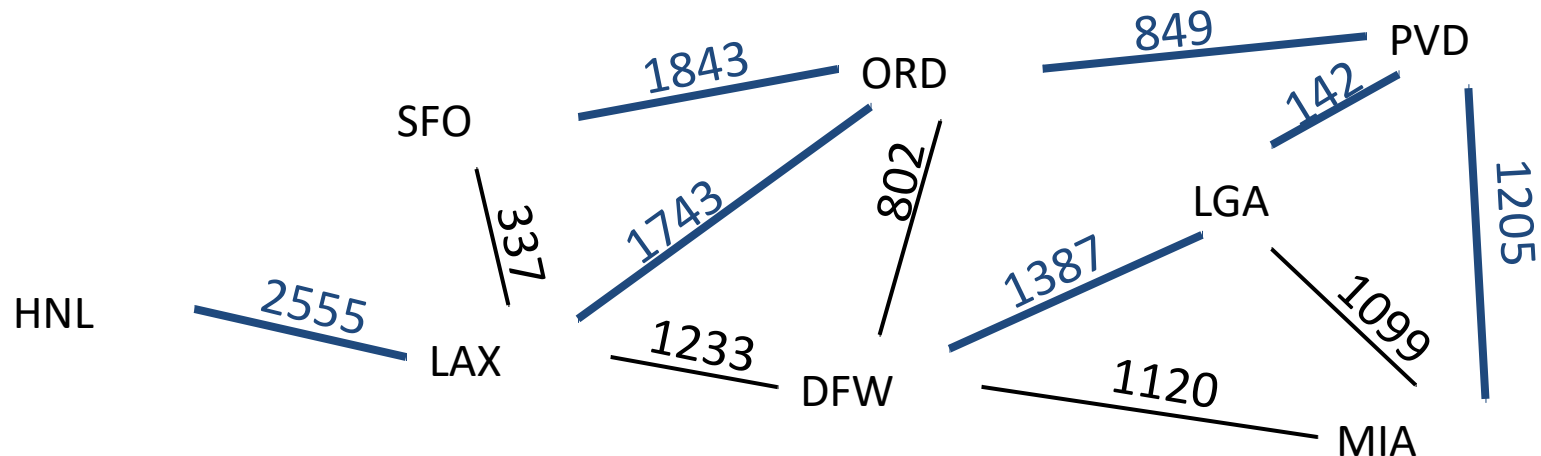
A subpath of a shortest path is itself a shortest path

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



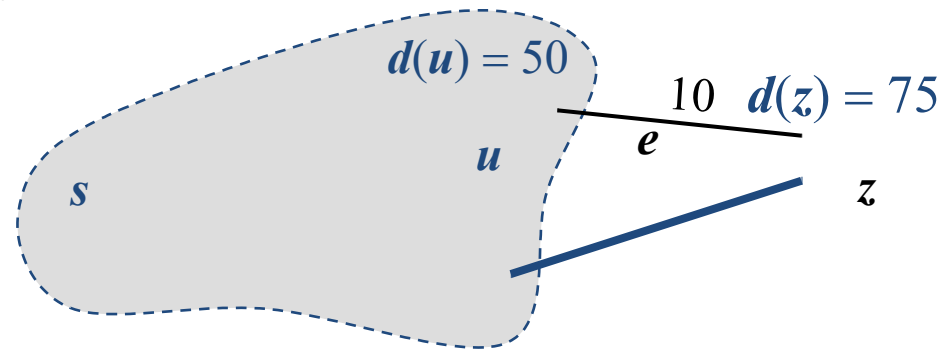
DIJKSTRA(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = G.V$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      for each vertex  $v \in G.Adj[u]$ 
8          RELAX( $u, v, w$ )
```

Edge Relaxation

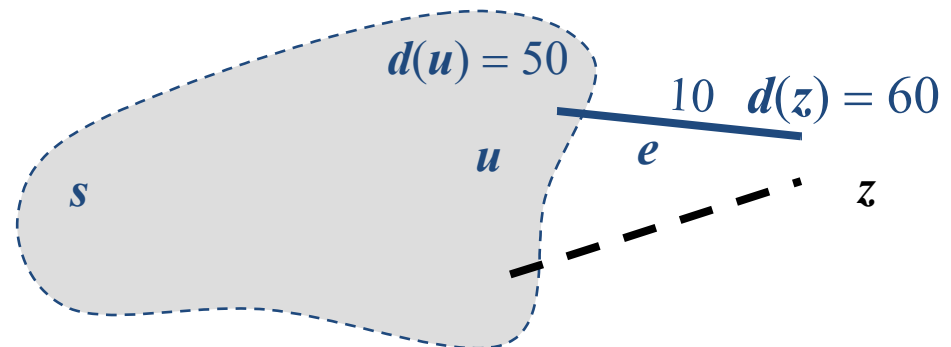


- Consider an edge $e = (u, z)$ such that
 - u is the vertex most recently added to the cloud
 - z is not in the cloud

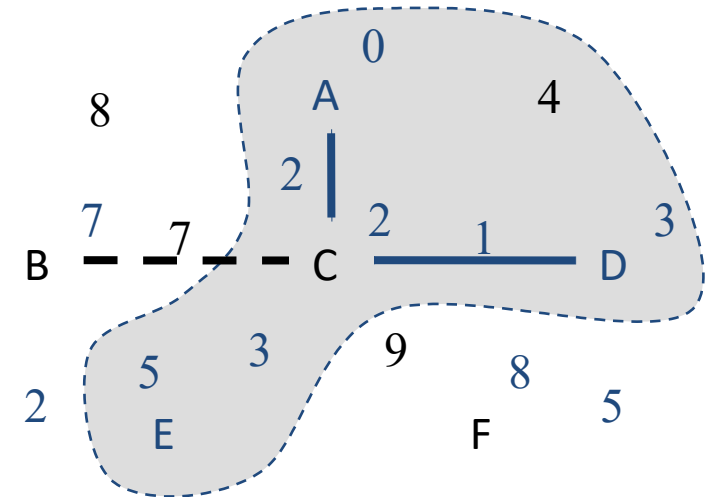
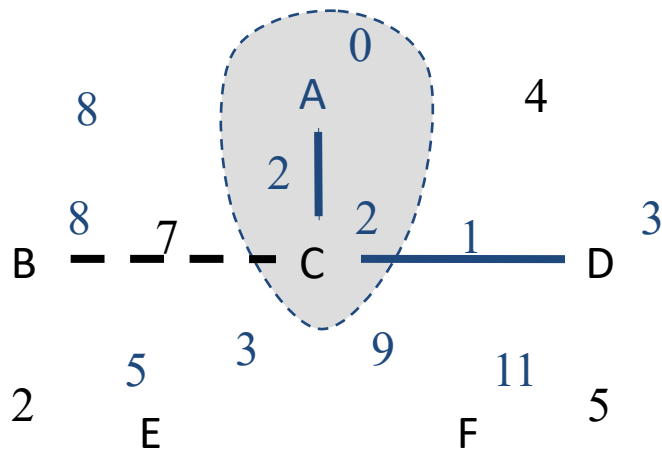
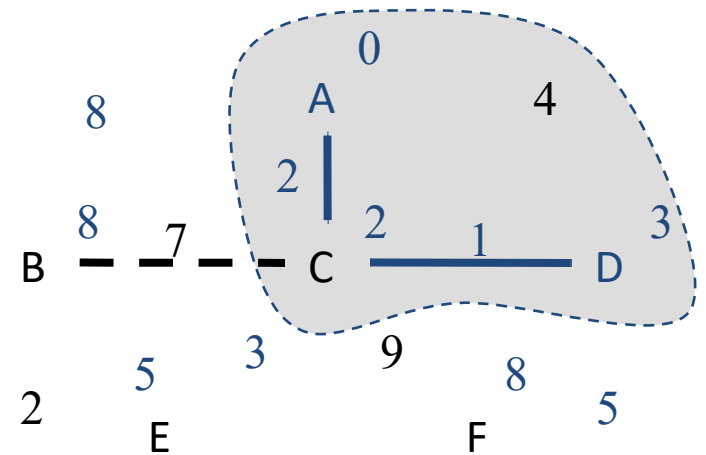
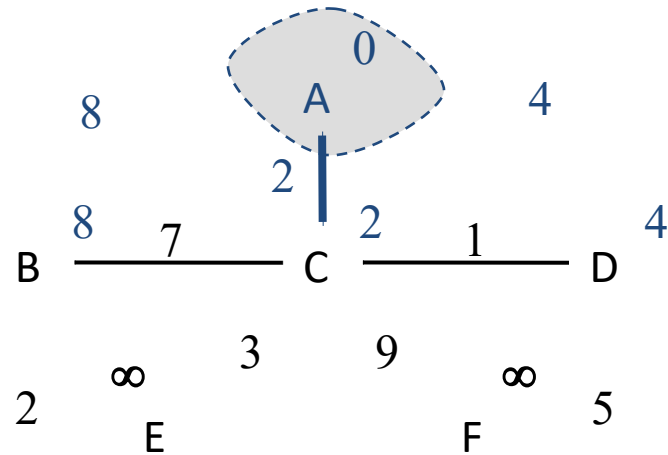


- The relaxation of edge e updates distance $d(z)$ as follows:

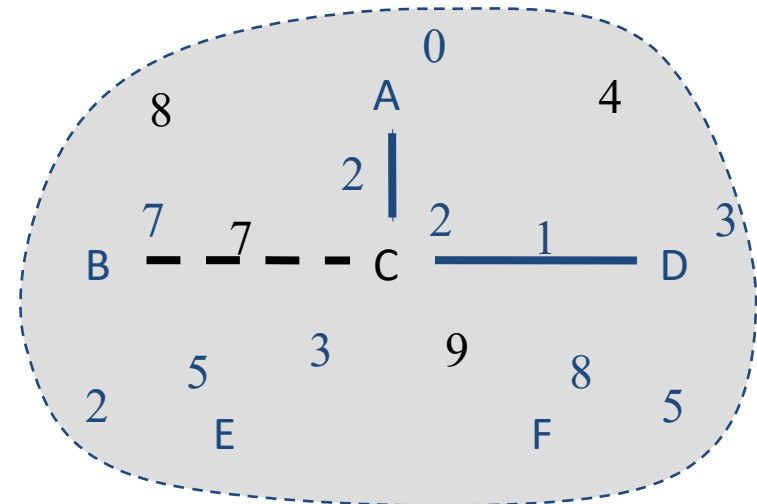
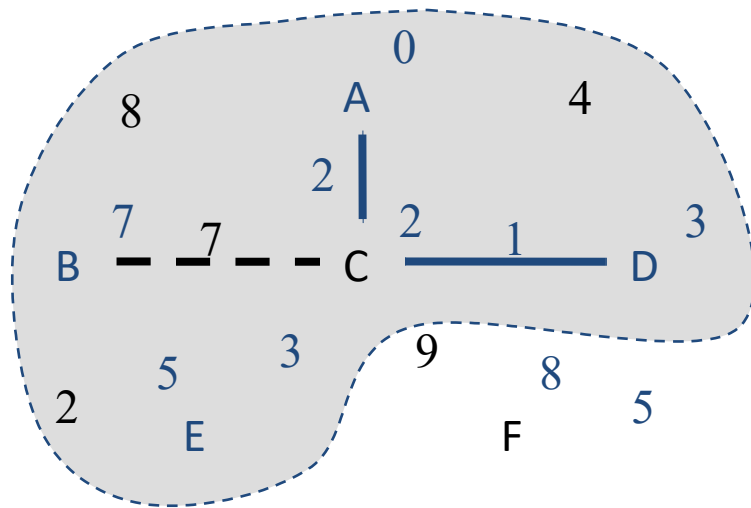
$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$



Example



Example (cont.)



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 two labels with each vertex:
 - Distance ($d(v)$ label)
 - locator in priority queue

Algorithm *DijkstraDistances*(G, s)

```
 $Q \leftarrow$  new heap-based priority queue
for all  $v \in G.vertices()$ 
    if  $v = s$ 
        setDistance( $v, 0$ )
    else
        setDistance( $v, \infty$ )
         $l \leftarrow Q.insert(getDistance(v), v)$ 
        setLocator( $v, l$ )
while  $\neg Q.isEmpty()$ 
     $u \leftarrow Q.removeMin()$ 
    for all  $e \in G.incidentEdges(u)$ 
        { relax edge  $e$  }
         $z \leftarrow G.opposite(u, e)$ 
         $r \leftarrow getDistance(u) + weight(e)$ 
        if  $r < getDistance(z)$ 
            setDistance( $z, r$ )

Q.replaceKey(getLocator(z), r)
```

- Graph operations
 - Method `incidentEdges` is called 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 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

- Using the template method pattern, we can extend Dijkstra's algorithm to return a tree of shortest paths from the start vertex to all other vertices
- We store with each vertex a third label:
 - parent edge in the shortest path tree
- In the edge relaxation step, we update the parent label

Algorithm *DijkstraShortestPathsTree*(G, s)

...

for all $v \in G.vertices()$

...

setParent(v, \emptyset)

...

for all $e \in G.incidentEdges(u)$

{ relax edge e }

$z \leftarrow G.opposite(u, e)$

$r \leftarrow getDistance(u) + weight(e)$

if $r < getDistance(z)$

setDistance(z, r)

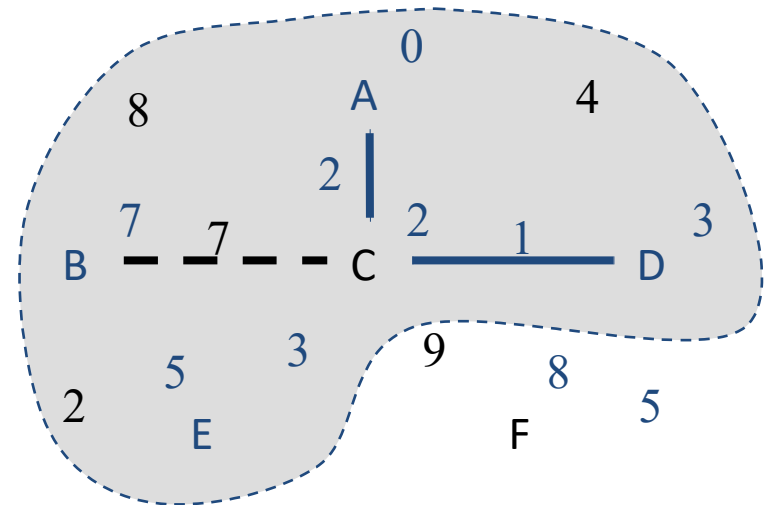
setParent(z, e)

Q.replaceKey(*getLocator*(z), r)

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 $d(F) \geq d(D)$, F's distance cannot be wrong. That is, there is no wrong vertex.

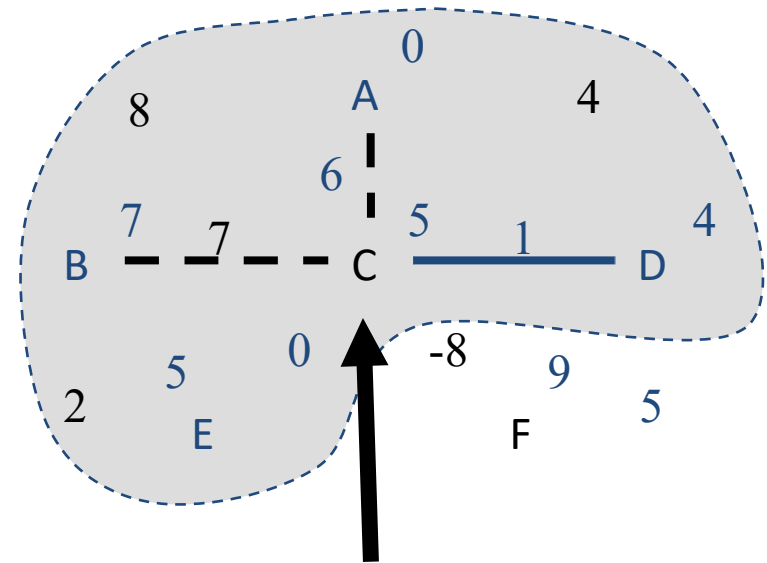


Why It Doesn't Work for Negative-Weight Edges



- ◆ 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



- 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?

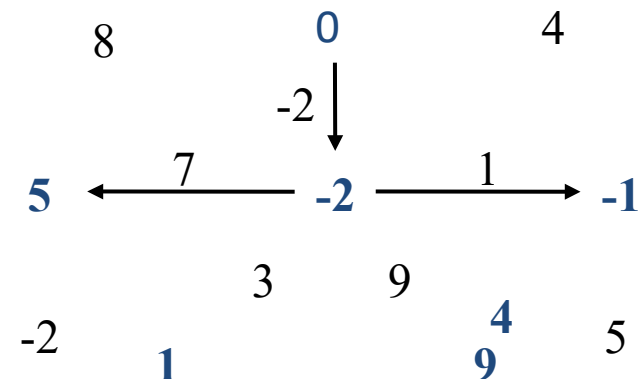
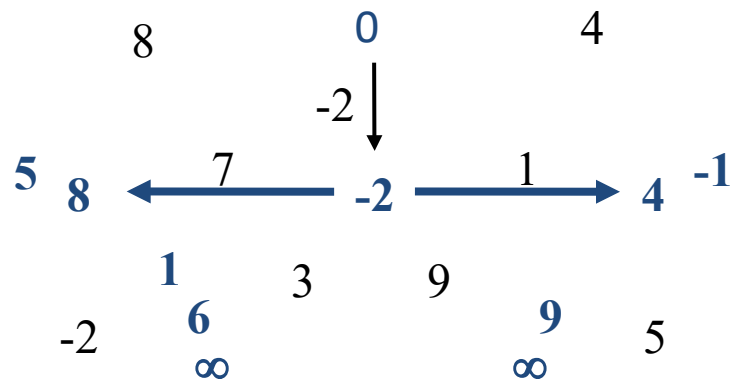
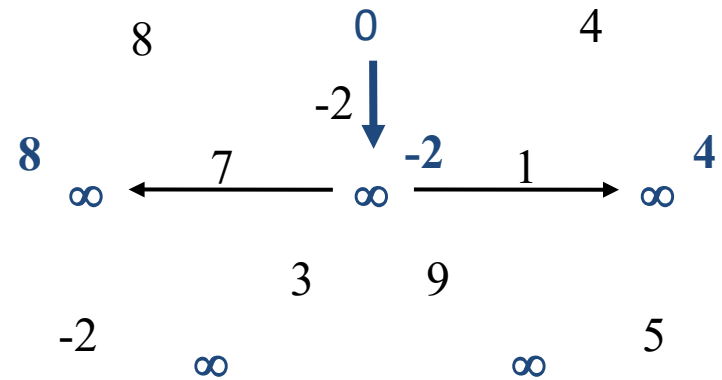
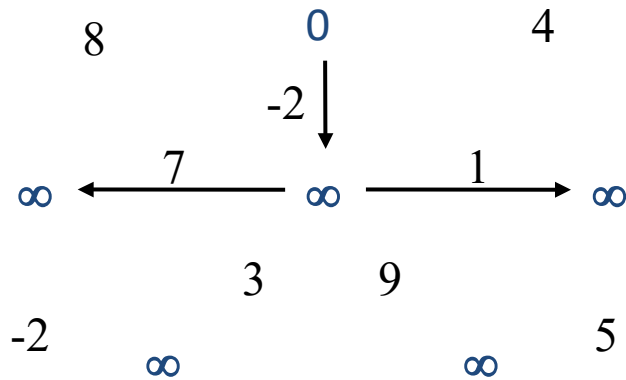
Algorithm *BellmanFord*(G, s)

```
for all  $v \in G.vertices()$ 
    if  $v = s$ 
        setDistance( $v, 0$ )
    else
        setDistance( $v, \infty$ )
for  $i \leftarrow 1$  to  $n-1$  do
    for each  $e \in G.edges()$ 
        { relax edge  $e$  }
         $u \leftarrow G.origin(e)$ 
         $z \leftarrow G.opposite(u, e)$ 
         $r \leftarrow getDistance(u) + weight(e)$ 
        if  $r < getDistance(z)$ 
            setDistance( $z, r$ )
```

Bellman-Ford 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 $O(nm \log n)$ 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 $1, \dots, n$ }

for all *vertex pairs* (i, j)

if $i = j$

$D_0[i, i] \leftarrow 0$

else if (i, j) is an edge in G

$D_0[i, j] \leftarrow \text{weight of edge } (i, j)$

else

$D_0[i, j] \leftarrow +\infty$

for $k \leftarrow 1$ to n **do**

for $i \leftarrow 1$ to n **do**

for $j \leftarrow 1$ to n **do**

$D_k[i, j] \leftarrow \min\{D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j]\}$

return D_n

Uses only vertices numbered $1, \dots, k$
(compute weight of this edge)

Uses only vertices
numbered $1, \dots, k-1$

k

Uses only vertices
numbered $1, \dots, k-1$

j



Thank You !!!