Shortest Paths

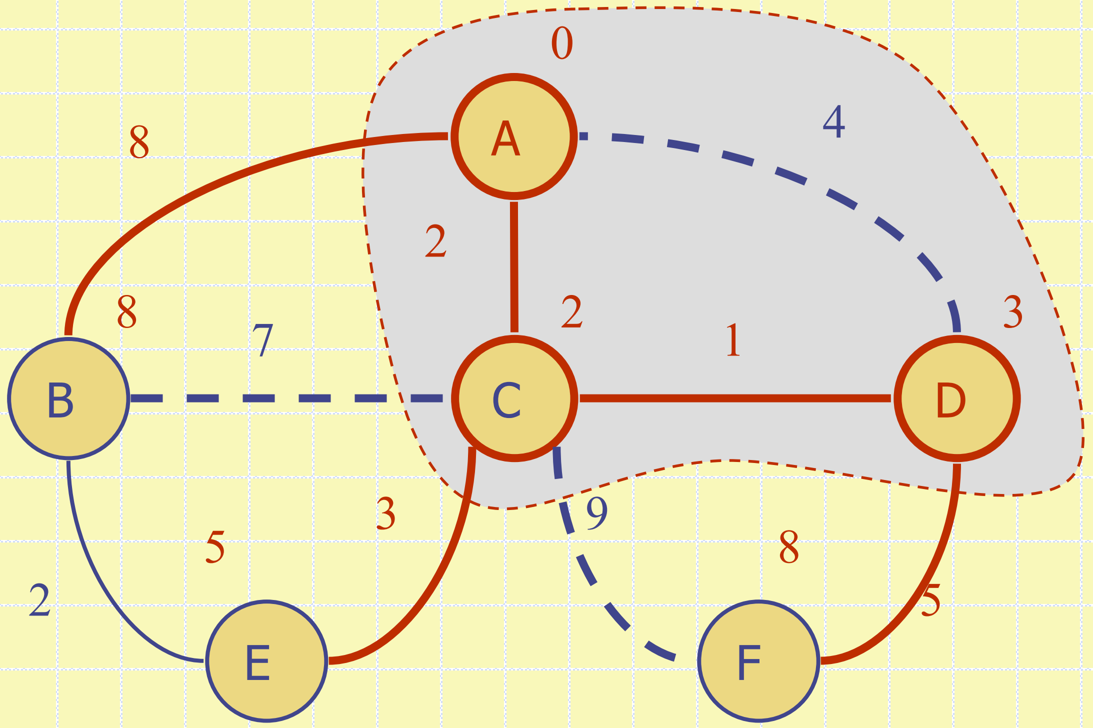
Dr. Aiman Hanna

Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

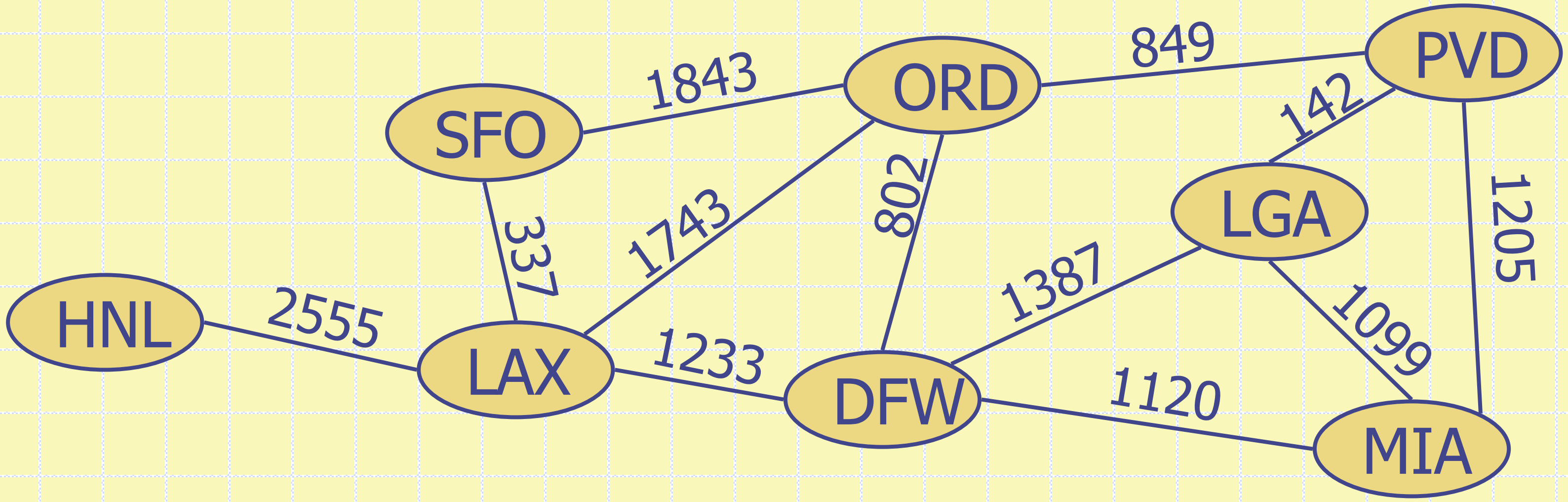
Coverage

- Shortest Paths



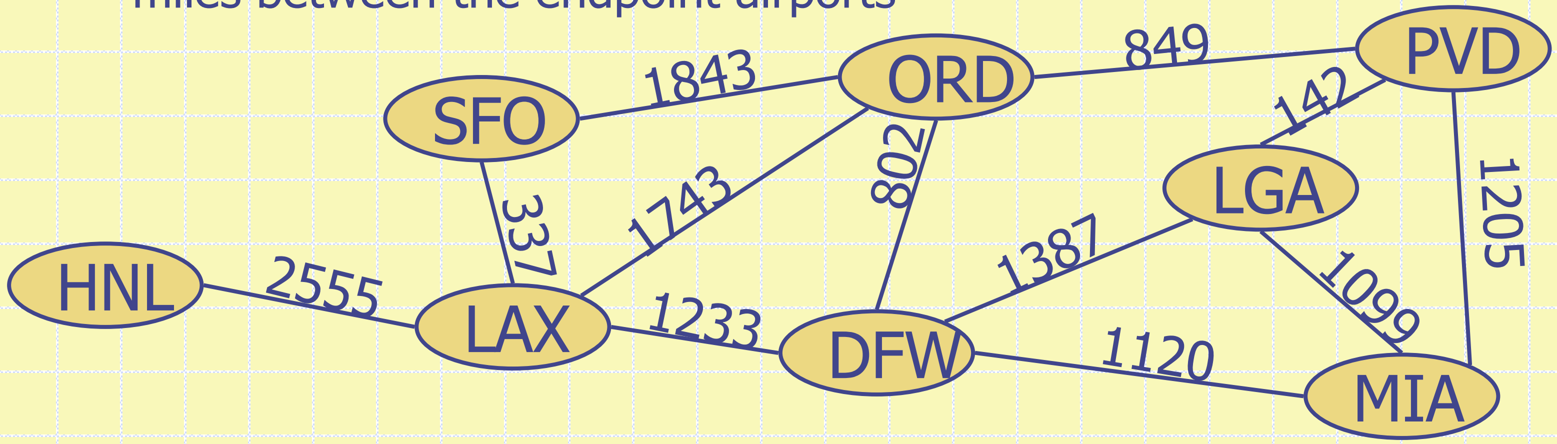
Weighted Graphs

- Breadth-first search can be used to find the shortest path from one starting vertex to all other ones assuming that all edges in the graph have the same weight (same length, cost, distance, etc.).
- However, in many situations this may not be the case, and hence BFS is inappropriate.



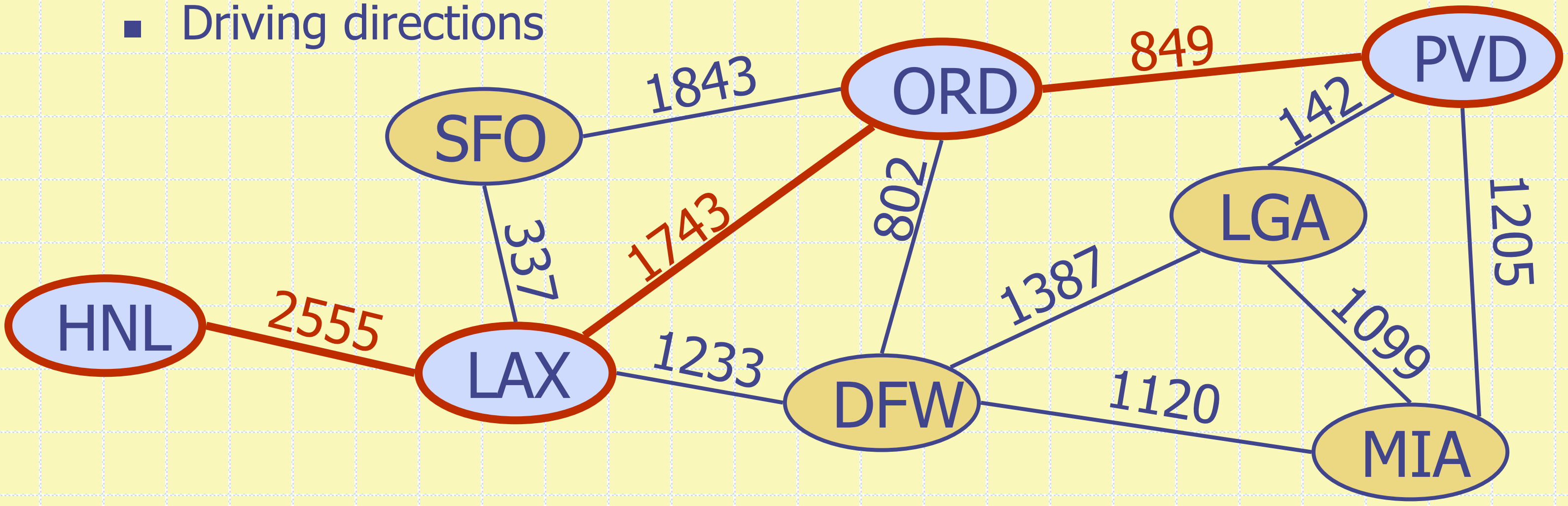
Weighted Graphs

- Graphs where edges have different ***weight***, are called ***weighted graphs***.
- In a weighted graph, each edge has an associated numerical value, called the weight of the edge, which can be different from one edge to another.
- Edge weight may represent, distances, costs, etc. For instance, 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:

The *length* (or weight) of a path is the sum of the weights of the edges composing that path

Property 2:

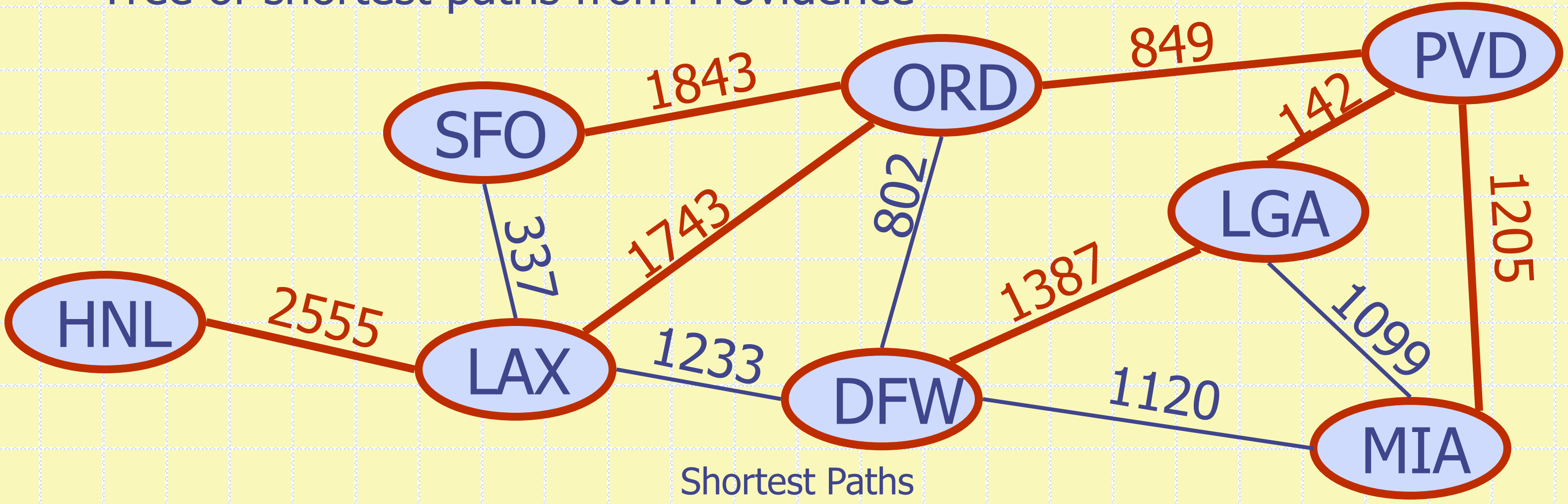
A subpath of a shortest path is itself a shortest path

Property 3:

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

Example:

Tree of shortest paths from Providence

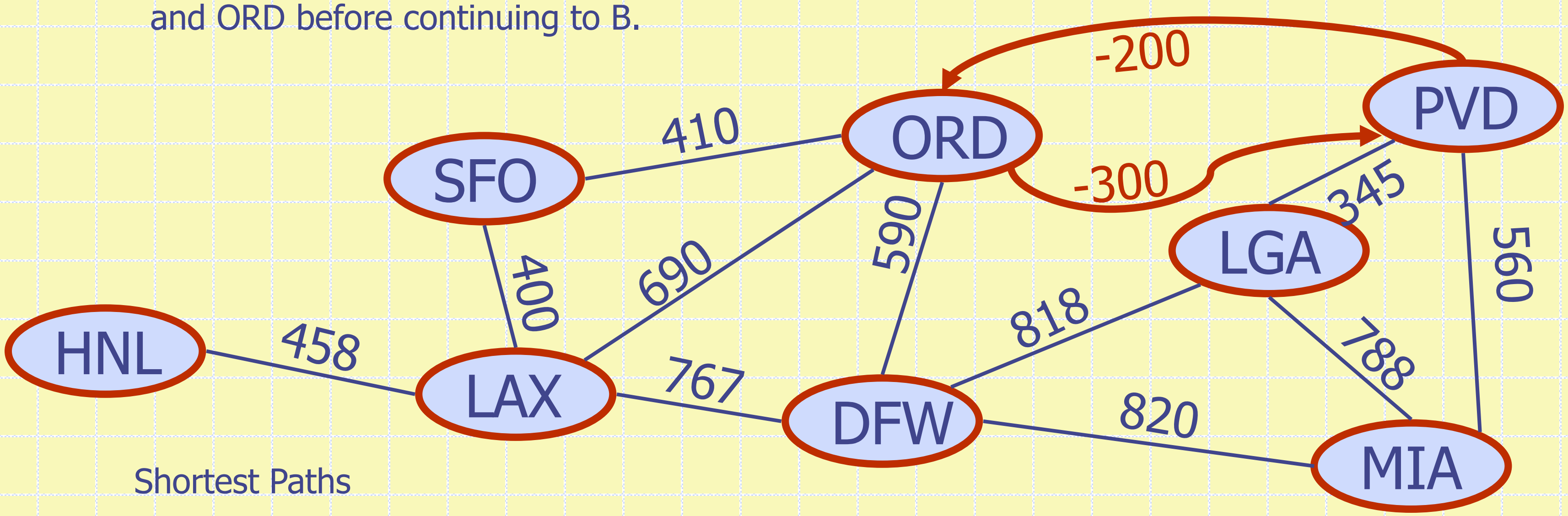


Negative-Weight Cycles

- The ***distance*** from vertex v to vertex u in a graph G, denoted by $d(v, u)$, is the length of the minimum length path (the shortest path) from v to u.
- $d(v, u) = +\infty$ if there is no path between v and u.
- There are cases however where the distance between v and u may not be defined even if there is a path form v to u in G. This is the case if the graph has a ***negative-weight cycle*** (a cycle whose total weight is negative).
- A ***negative-weight edge*** is an edge whose weight is negative.
- For instance, assume that the edges in the airport graphs have weights that define the cost to travel between these cities. Further, assume that someone has offered to pay travelers an amount that is larger than the actual cost to travel between PVD and ORD.

Negative-Weight Cycles

- Consequently, the cost from PVD to ORD (that edge *e = (PVD, ORD)*) would carry a negative value, which makes *e = (PVD, ORD)* a negative-weight edge.
- Further, assume that someone else has then offered travelers to pay an amount that is larger than the actual cost to travel between ORD and PVD.
- Consequently, there is now a **negative-weight cycle** (cycle with a total weight that is negative) between PVD and ORD. These cycles are very undesirable and must be avoided when edge weights are used to represent distances.
- Now, the distances of that graph can no longer be defined since someone can build a path from any city *A* to another *B* going through PVD and making as many cycles as desired between PVD and ORD before continuing to B.



Dijkstra’s Algorithm

- Dijkstra’s algorithm computes the distances (the shortest paths) of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative

Dijkstra’s Algorithm

- **Edge Relaxation**:
  - Let us define a label ***D[u]*** for each vertex u in the graph, where $D[u]$ stores an approximation of the shortest path distance from a starting vertex v to u.
  - In other words, $D[u]$ will always store the length of the best path to u that could have been found **so far**.
  - Initially, $D[v] = 0$, and $D[u] = +\infty$ for each other vertex u in the graph.
  - $D[u]$ will afterwards be updated if a better path is found that results in a smaller shortest path value to u.
  - The operation is known as ***relaxation***; the idea is similar to stretching a spring more than needed then “relax” it back to its true resting distance.

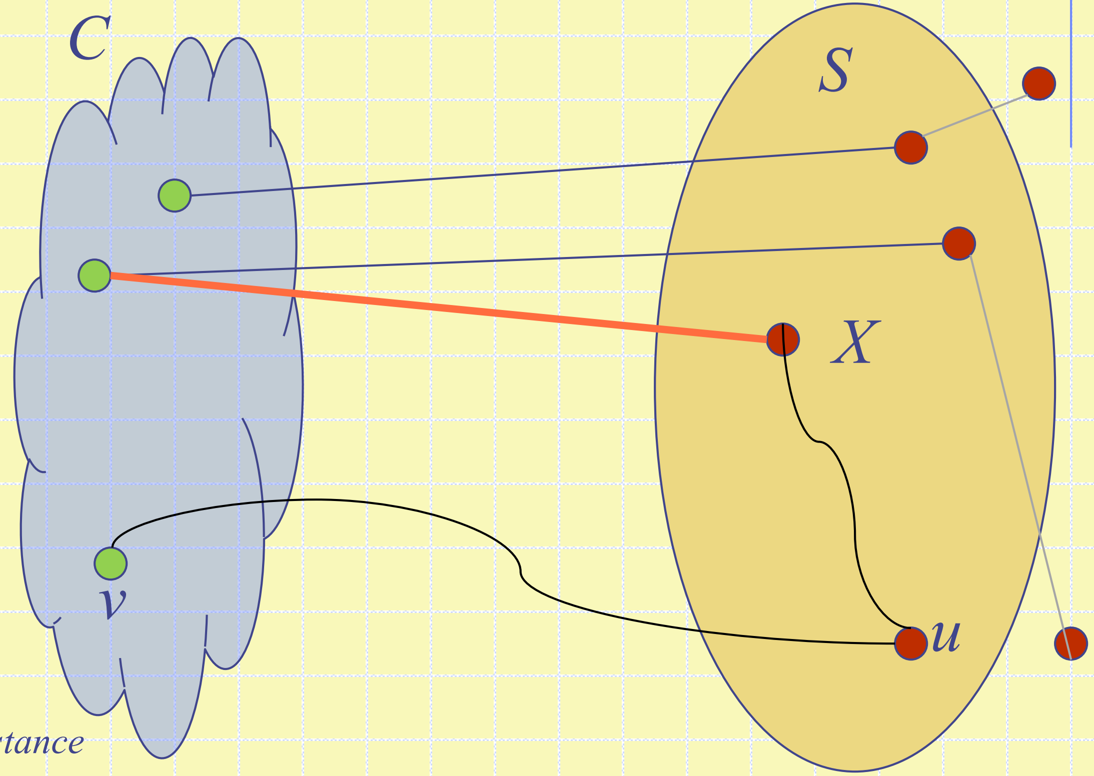
Dijkstra’s Algorithm

- The idea is to grow a “cloud” of vertices, beginning with the stating vertex ***s*** and eventually covering all the vertices
- In details, the algorithm starts by defining two sets; let us refer to them as the C (for Cloud) set and the *S* set.
- Initially, *C* is empty, while *S* has vertex ***s***. Additionally, all the distance approximations, ***D[u]***, as initialized as ∞,with the exception of ***D[v]***, which is initialized as 0.
- Afterwards *C* will grow to include specific vertices one at a time (those are the ones with concrete shortest paths), while *S* will include all other vertices that are directly connected either to ***s*** or to other vertices already in *C*. Algorithm terminates once all vertices have been moved to *C*.

Dijkstra's Algorithm (continues…)

- In details, the algorithm performs the repeated moving operations from S to C as follows:
  - Move one vertex from S to C. Let us refer to that vertex as X. This vertex, X, must satisfy the following:
    1) It is either directly connected to v or directly connected to another vertex that is already in C
    2) Has the smallest distance to v among all the potential vertices that are considered for movement to C
  - Whenever the vertex X is moved to C, its distance is the smallest distance (shortest path) to v
  - Additionally, once this vertex X is chosen, move it to C and recheck all distances of all vertices that have direct connection to X and are NOT yet in C. If a smaller value than what we already have is found, update this value
  - In each step, we also update how to go to these vertices (that is, through which vertex); we refer to this vertex as the ***prior function***
  - The steps are repeated until all the graph vertices are moved to C. The final obtained distances represent the shortest paths, and the prior functions indicate the direction from v (through which vertex) to obtain these shortest paths to each of the other vertices in the graph

Dijkstra's Algorithm (continues…)



─	*Distances yet to be determined*
─	*Distances to potential vertices*
━	*Potential vertex with shortest distance*
X	*Selected vertex to move to C*
∽	*Different routes to u.*

Is there a cheaper route through X to other vertices in S? If so, update distances to these vertices.

Dijkstra’s Algorithm

- Example: What are the shortest paths from A to other vertices?*

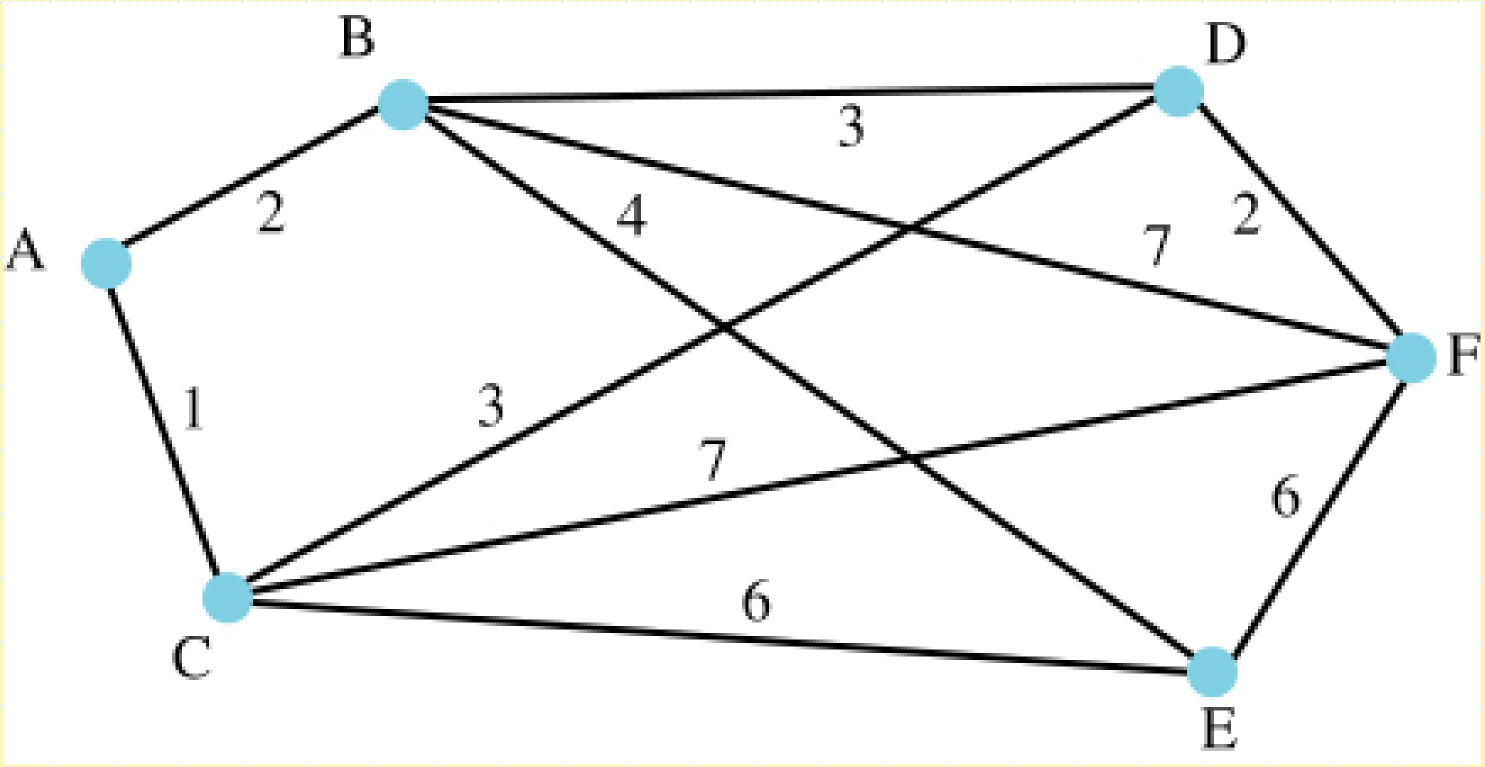


*Figure is © *Understanding Data Communications and Networks* by William A. Shay - PWS publishing company, Third Edition. **ISBN-10:** 0534383173, **ISBN-13:** 978-0534383176.

Dijkstra's Algorithm

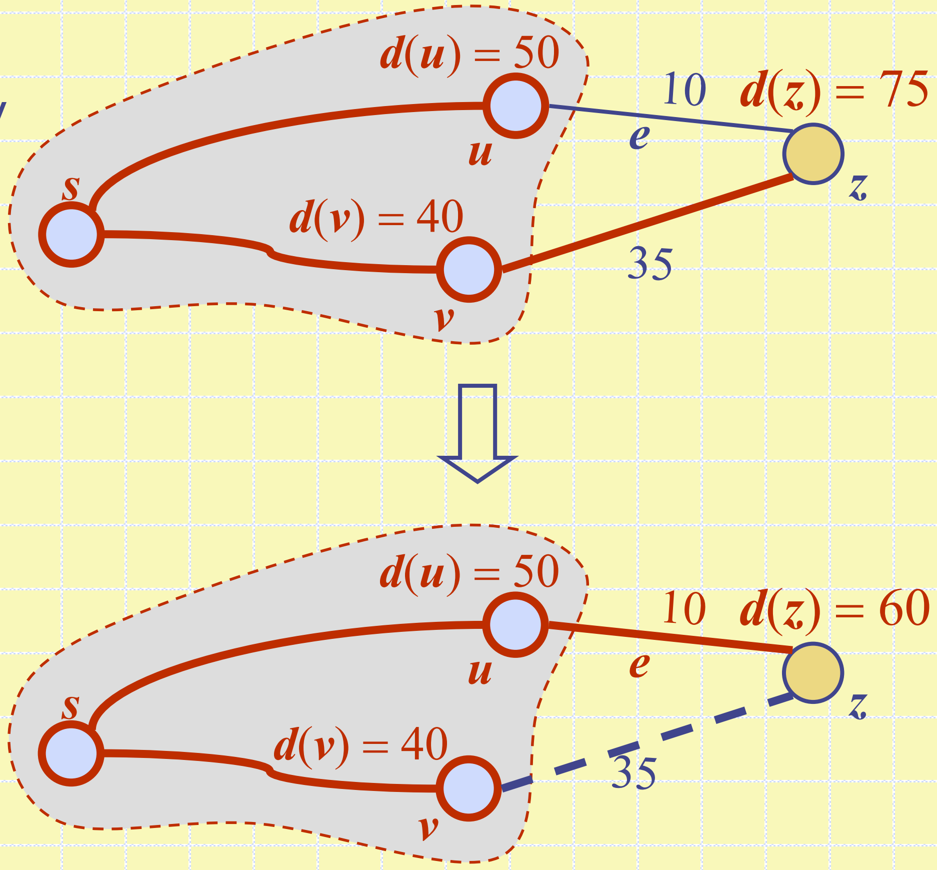
				Cost function						Prior function					
	C	Potential elements of S	**X**	A	B	C	D	E	F	A	B	C	D	E	F
0	{}	{A}	**A**	**0**	∞	∞	∞	∞	∞	**A**	-	-	-	-	-
1	{A}	{B,C}	**C**	0	2	**1**	∞	∞	∞	A	A	**A**	-	-	-
2	{A, C}	{B,D,E,F}	**B**	0	**2**	1	4	7	8	A	**A**	A	C	C	C
3	{A,B,C}	{D,E,F}	**D**	0	2	1	**4**	6	8	A	A	A	**C**	*B*	C
4	{A,B,C,D}	{E,F}	**E**	0	2	1	4	**6**	6	A	A	A	C	**B**	*D*
5	{A,B,C,D,E}	{F}	**F**	0	2	1	4	6	**6**	A	A	A	C	B	**D**

Note: See comments attached to this slide for detailed information on how the operations progressed.

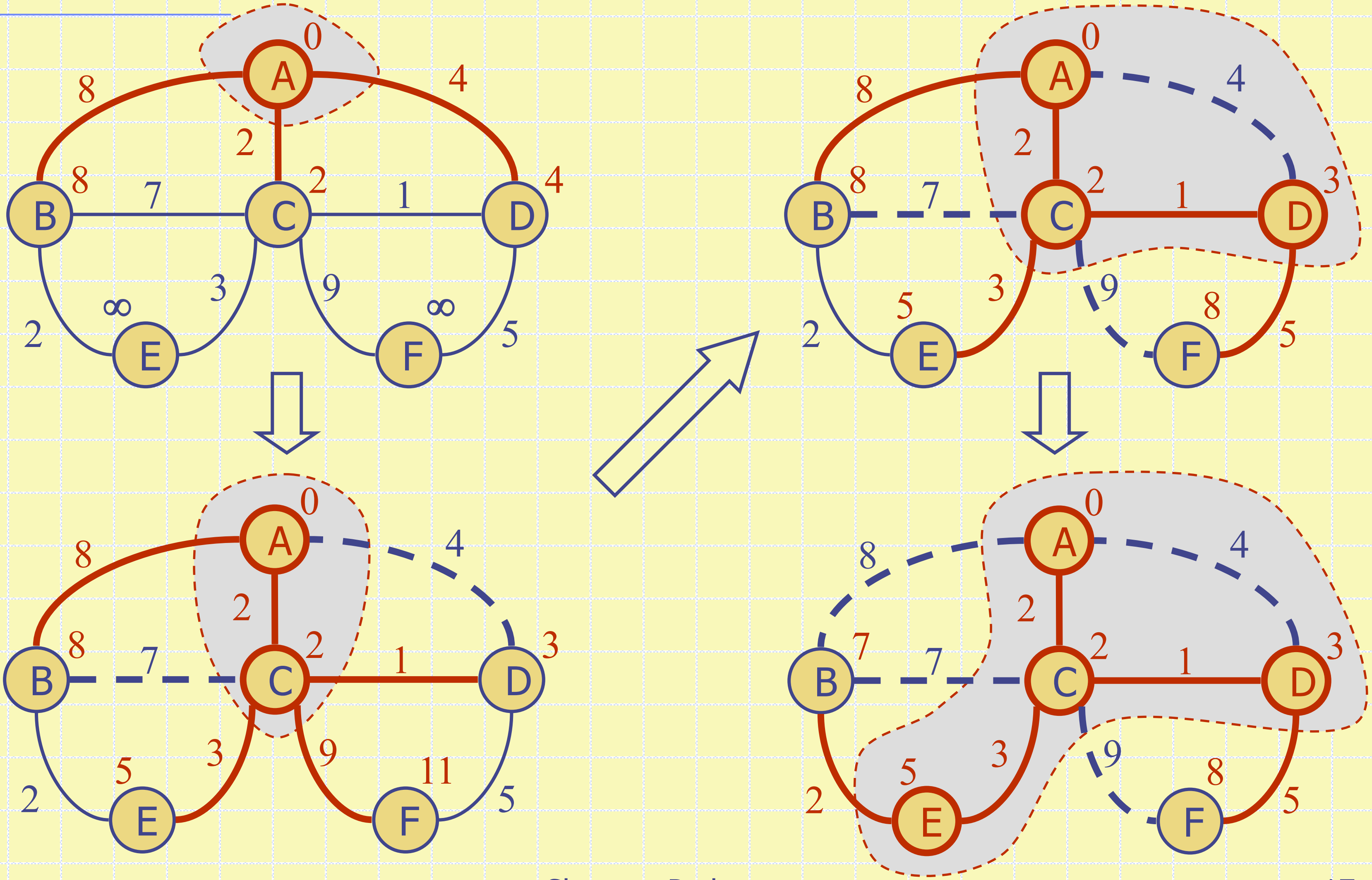
Edge Relaxation

- 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:

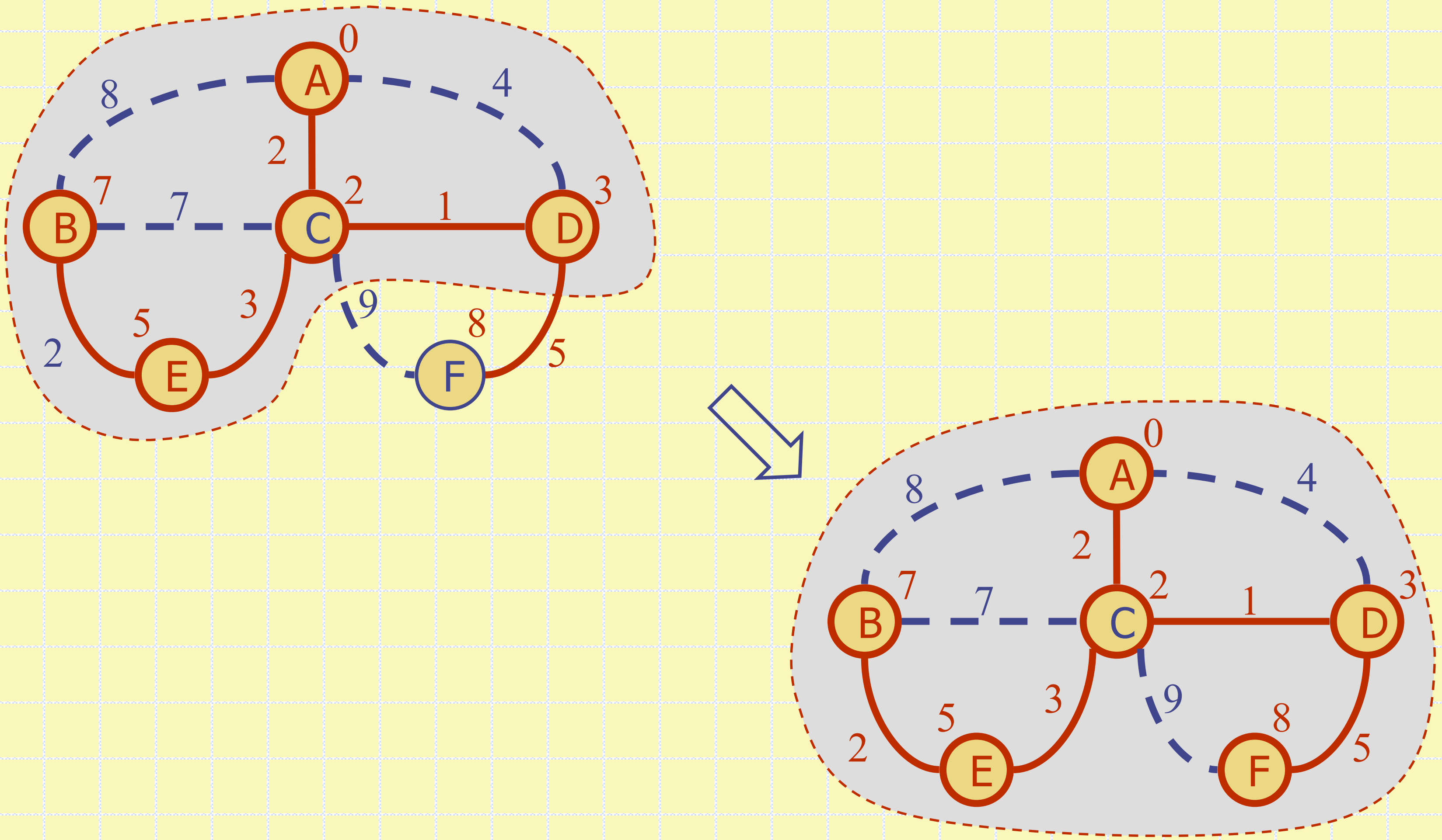
$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



Example 2 – Visual Approach



Example 2 (cont.)



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*
- We store two labels with each vertex:
  - Distance
  - Entry in priority queue

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

Analysis of Dijkstra's Algorithm

- Initial operations (the 1st **for all loop**)
  - Method *setDistance* is called once for each vertex, resulting in $O(n)$
  - Method *setEntry* is called once for each vertex, resulting in $O(n)$
  - Each vertex is then inserted into the Heap-based Priority Queue, where each insertion costs $O(\log n)$, resulting in a total of $O(n \log n)$
- Removal from Priority queue (the while loop)
  - Each vertex is removed once from the priority queue, where each removal takes $O(\log n)$ time, resulting in a total of $O(n \log n)$
  - Method *getValue* is called once for each vertex, resulting in $O(n)$

➔ so far, complexity is $O(2\, n \log n)$ ➔ **$O(n \log n)$**

Analysis of Dijkstra's Algorithm

- Following all incident edges of removed vertex (the inner **for loop**)

    - Now, the method *incidentEdges* is called for every removed vertex *u*, which means that the method is called as many times as the degree of that vertex *u*, resulting in $O(\deg(u))$, provided the graph is represented by the adjacency list structure.

    - Since all vertices are removed one by one, and sicne each will cost $O(\deg(v))$ of that vertex *v*, a total of $O(\Sigma_v \deg(v))$ for all vertices is eventually incurred.

    - What happens however, each time you follow one of the incident edges of *u* ?

Analysis of Dijkstra’s Algorithm

- What happens however, each time you follow one of the incident edges of *u* ?
    - The methods: *opposite*, *getDistance*, *weight*, and *setDistance* are called, however each of these cost $O(1)$
    - **BUT ……**
    - The method ***replaceKey***, which affects the values of the key in the heap, may result in $O(\log n)$ operations since the heap may need to be corrected each time
- This consequently results in $O(\log n)$ for **each** incident edge
- Since *incidentEdges* method is eventually called $O(\Sigma_v \deg(v))$, since all vertices are removed, a total complexity of $O(\Sigma_v \deg(v) \log n)$ is incurred

Analysis of Dijkstra's Algorithm

- So, what is the total complexity?
  - From initial operations and insertion into the queue, we have: $O(n \log n)$

- From following all the incident edges, we have $O(\Sigma_v \deg(v) \log n)$

- Recall that $\Sigma_v \deg(v) = 2m$
  - $O(\Sigma_v \deg(v) \log n) = O(2m \log n) \rightarrow O(m \log n)$

- So, total complexity of Dijkstra's algorithm is:
- $O(n \log n) + O(m \log n) \rightarrow O((n + m) \log n)$

Analysis of Dijkstra's Algorithm

- In conclusion, Dijkstra's algorithm runs in

  $$O((n + m) \log n)$$

  provided the graph is represented by the adjacency list structure.

- The running time can also be expressed as a function of n only as

  $$O(n^2 \log n)$$

  - Recall that $m <= n(n-1)/2$

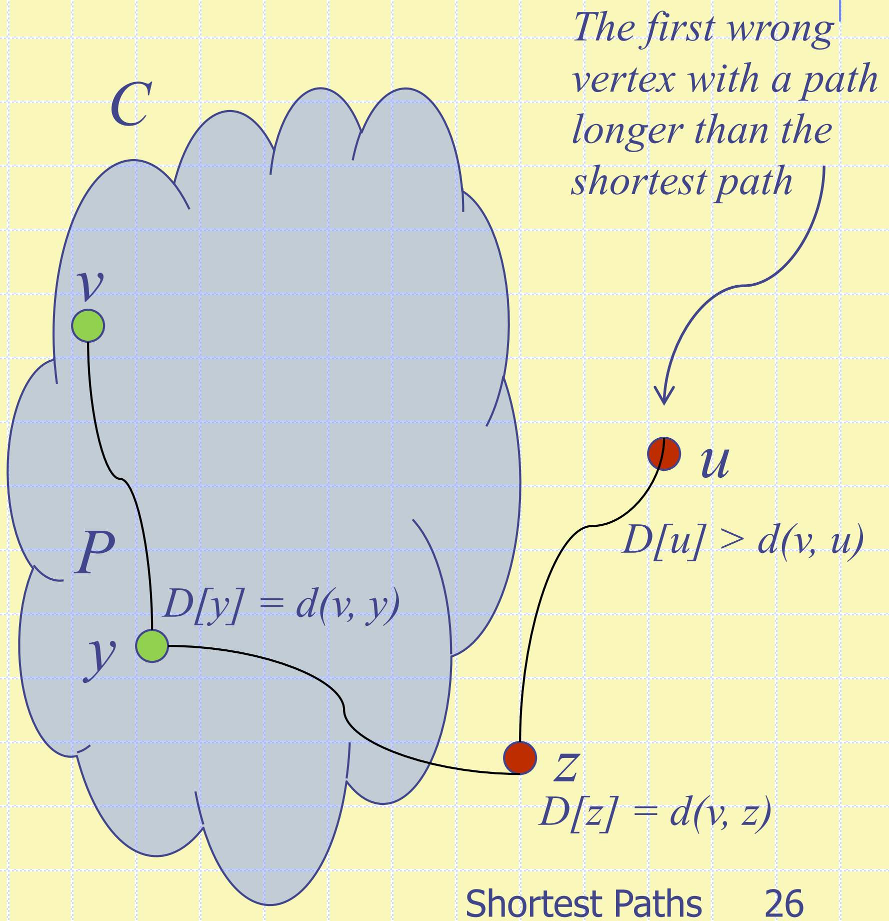
Shortest Paths Tree

- 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 ∈ G.vertices()
    …
    setParent(v, ∅)
    …

    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)
        setParent(z,e)
        Q.replaceKey(getEntry(z),r)
```

Why Dijkstra's Algorithm Works

- Dijkstra's algorithm is based on a method called the ***greedy method***. It adds vertices by increasing distance.
  - Assume that vertex u is the ***first*** vertex that was incorrectly picked up to enter the cloud
  - By that time, both v (the starting vertex), and y (a correctly picked up vertex) are already in the cloud
  - However, assume that the algorithm misbehaves such that it has incorrectly picked up u to enter the cloud without having a correct shortest path



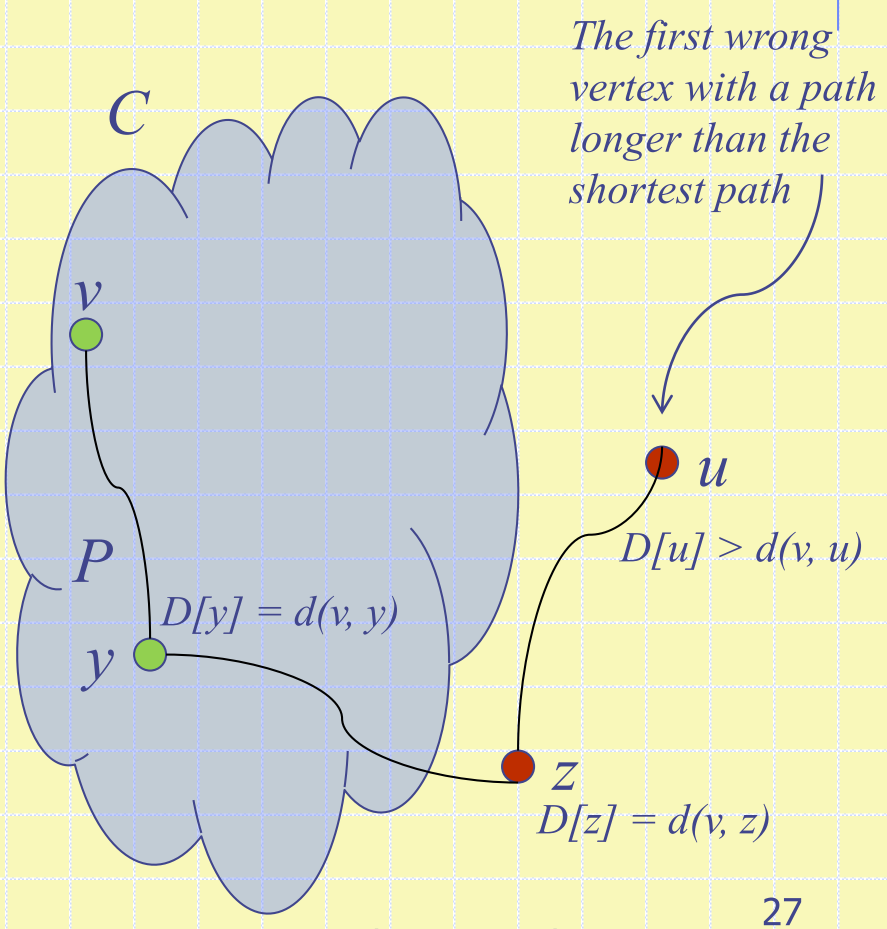
The first wrong vertex with a path longer than the shortest path
C
v
P
$D[y] = d(v, y)$
y
u
$D[u] > d(v, u)$
z
$D[z] = d(v, z)$

Why Dijkstra’s Algorithm Works

- Since u was chosen by error, then $\boldsymbol{D[u] > d(v, u)}$
- Let then back a bit on time to the moment when y was added to the cloud.
- At that time, we must have evaluated the relaxed distance to z, $D[z]$, and possibly have updated it to reflect the current known shortest path to z.
- Consequently, at this point we had:

  $\boldsymbol{D[z] <= D[y] + w((y, z))}$

  $\boldsymbol{\Rightarrow D[z] <= d(v, y)+ w((y, z))}$


The first wrong vertex with a path longer than the shortest path
C
v
P
D[y] = d(v, y)
y
u
D[u] > d(v, u)
z
D[z] = d(v, z)

Why Dijkstra's Algorithm Works

- But, since z is indeed having a correct shortest path (recall that u is he first incorrect vertex) then

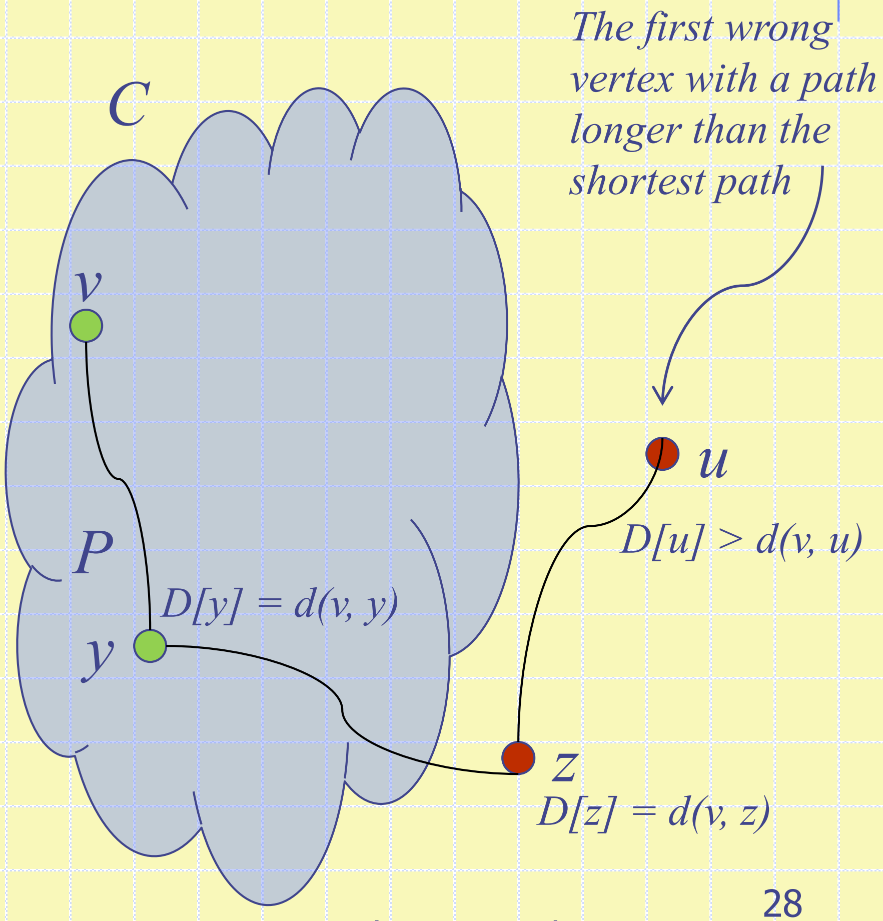
  $$D[z] = d(v, z)$$

- But since now the algorithm is picking up u not z, it must have found that:

  $$D[u] <= D[z]$$

- Recall that a shortest path is composed of shortest sub-paths. Hence, since z is in the shortest path, P, from v to u then:

  $$d(v, z)+ d(z, u) = d(v, u)$$



The first wrong vertex with a path longer than the shortest path
C
v
u
$D[u] > d(v, u)$
P
$D[y] = d(v, y)$
y
z
$D[z] = d(v, z)$

Why Dijkstra's Algorithm Works

- Moreover, $d(z, u) > 0$ since there are no negative edges

- Therefore:

  $D[u] <= D[z] = d(v, z)$

  ➔ $d(v, z) <= d(v, z) + d(z, u)$

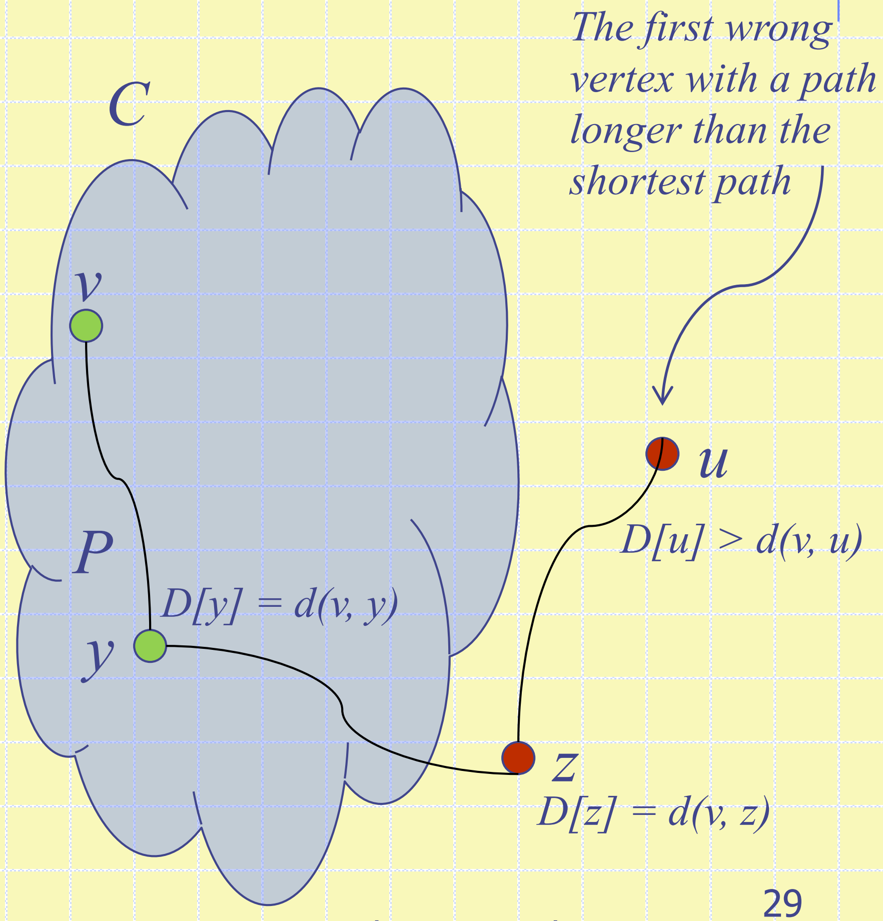
  ➔ $d(v, z) <= d(v, u)$

  ➔ $D[u] <= d(v, u)$

- However this contradicts our original definition of u and assumption that

  $D[u] > d(v, u)$

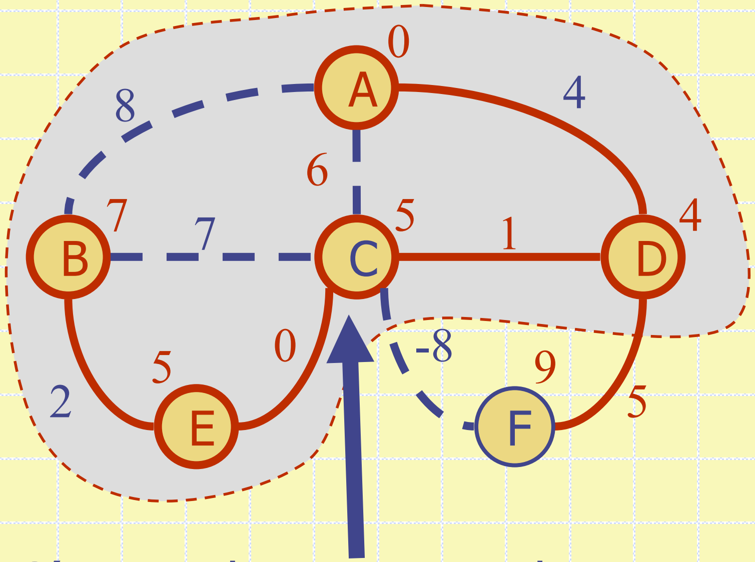
➔ Which means that there can never be such a vertex u.


The first wrong vertex with a path longer than the shortest path
C
v
P
D[y] = d(v, y)
y
u
D[u] > d(v, u)
z
D[z] = d(v, z)

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 and possibly has already affected many other distances!

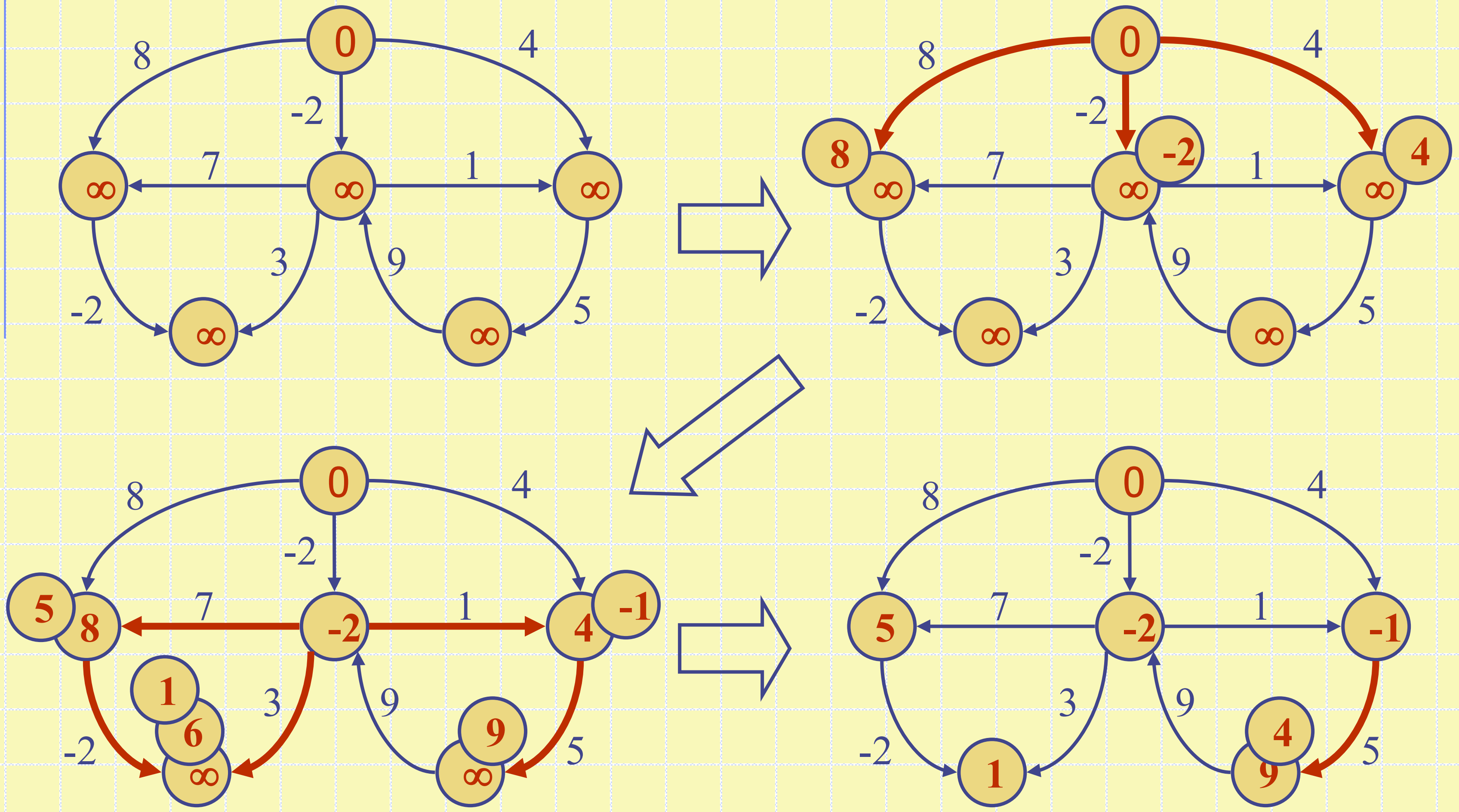
Bellman-Ford Algorithm (not in book)

- 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  $v \in G.vertices()$
        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



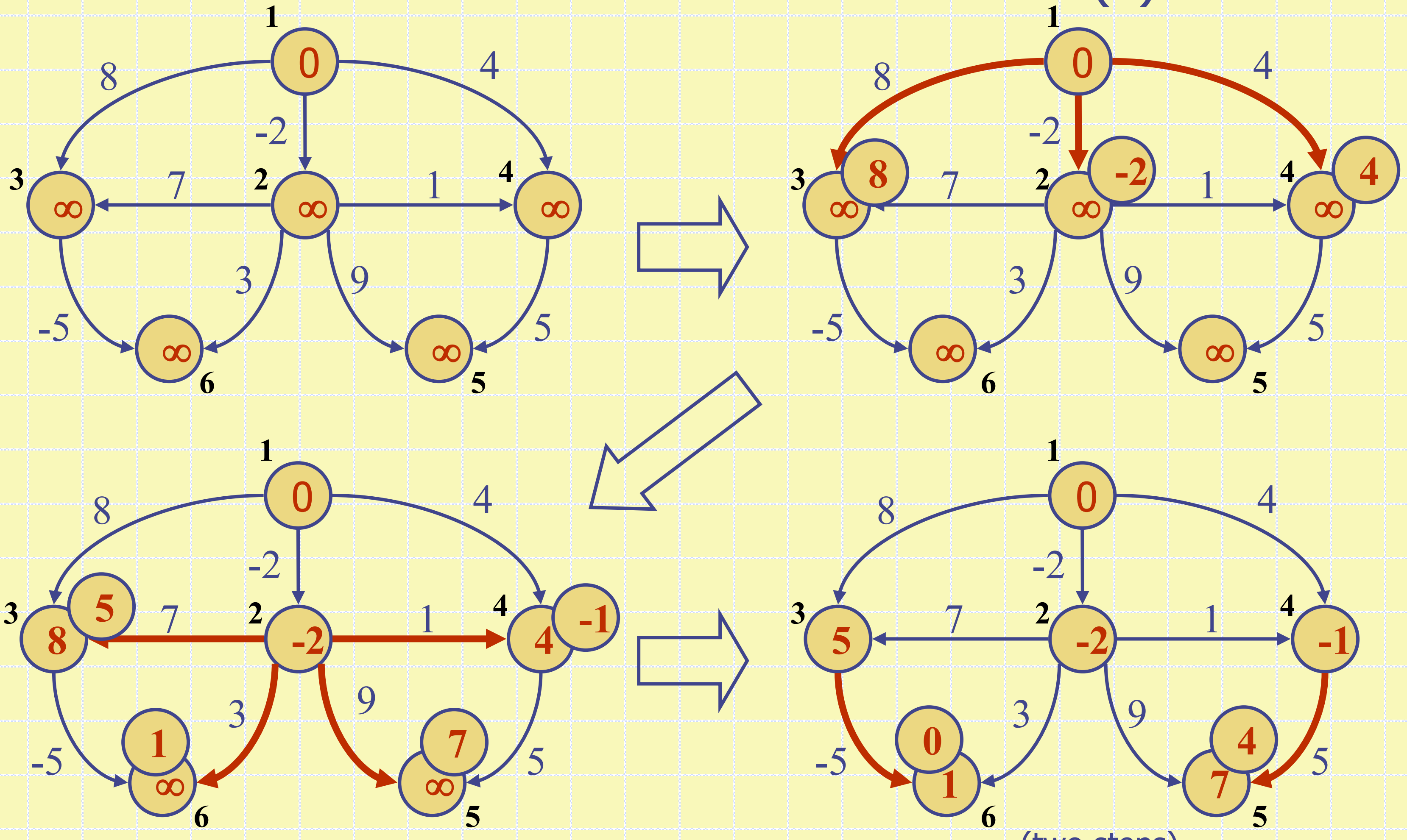
DAG-based Algorithm (not in book)

- 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



1
0
8
4
-2
3
2
4
7
1
∞
∞
∞
3
9
-5
5
∞
∞
6
5
8
-2
4
5
-1
1
7
0
1
4
7
(two steps)