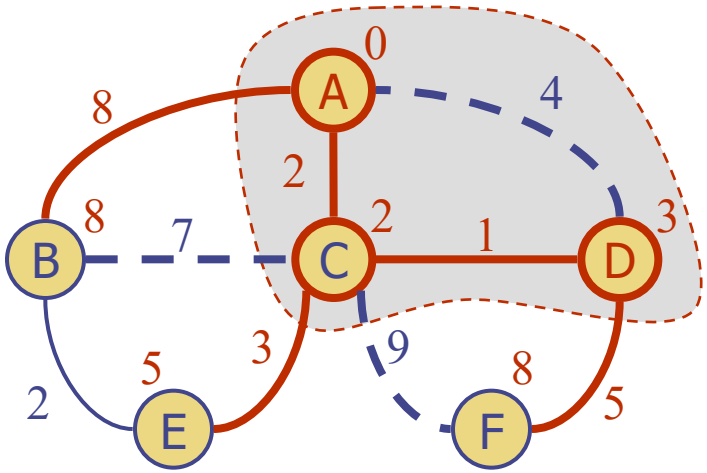
Graph and Shortest Paths

Algorithms & Data Structures
ITCS 6114/8114

Dr. Dewan Tanvir Ahmed
Department of Computer Science
University of North Carolina at Charlotte

Shortest Paths



Outline and Reading

- Graph Facts
- Weighted graphs
  - Shortest path problem
  - Shortest path properties
- Dijkstra's algorithm
  - Algorithm
  - Edge relaxation
  - Example
  - Analysis

Graphs

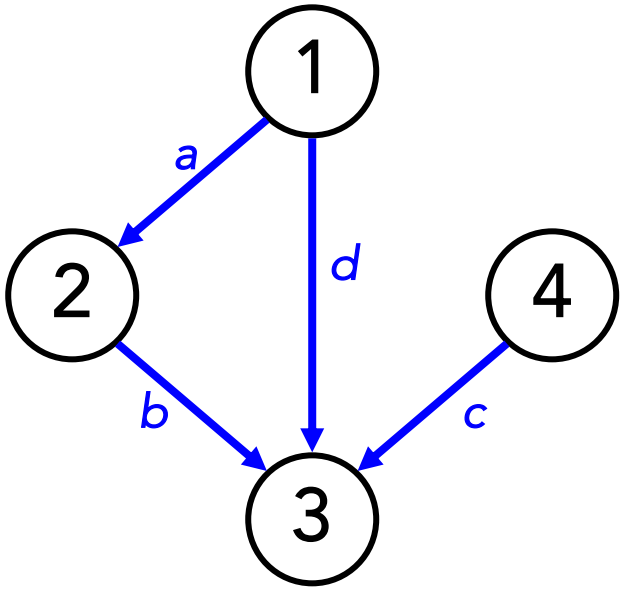
- A graph G = (V, E)
  - V = set of vertices
  - E = set of edges = subset of V × V
  - Thus |E| = O(|V|^2)

- We will typically express running times in terms of $|E|$ and $|V|$ (often dropping the |'s)
  - Dense - If $|E| \approx |V|^2$
  - Sparse - If $|E| \approx |V|$

- If you know you are dealing with dense or sparse graphs, different data structures may make sense

Representing Graphs

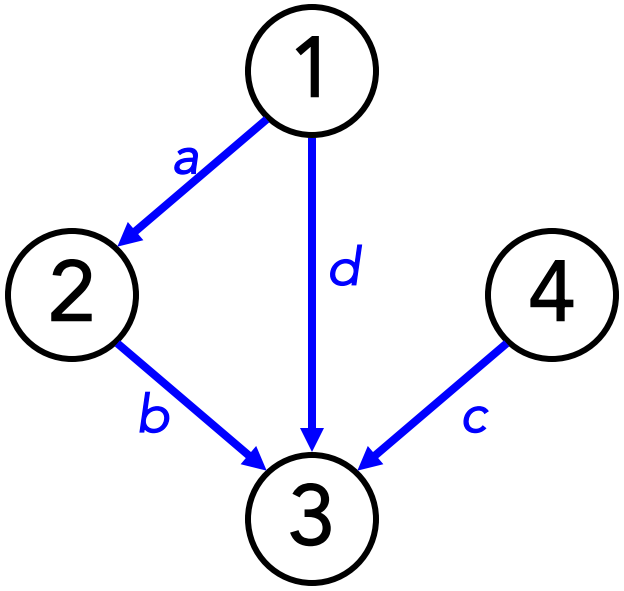
- Assume $V = \{1, 2, \ldots, n\}$
- An *adjacency matrix* represents the graph as a $n \times n$ matrix A:
  - $A[i, j]$ = 1 if edge $(i, j) \in E$ (or weight of edge)
    = 0 if edge $(i, j) \notin E$



A	1	2	3	4
1				
2				
3				
4				

Graphs: Adjacency Matrix

- Example:



A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

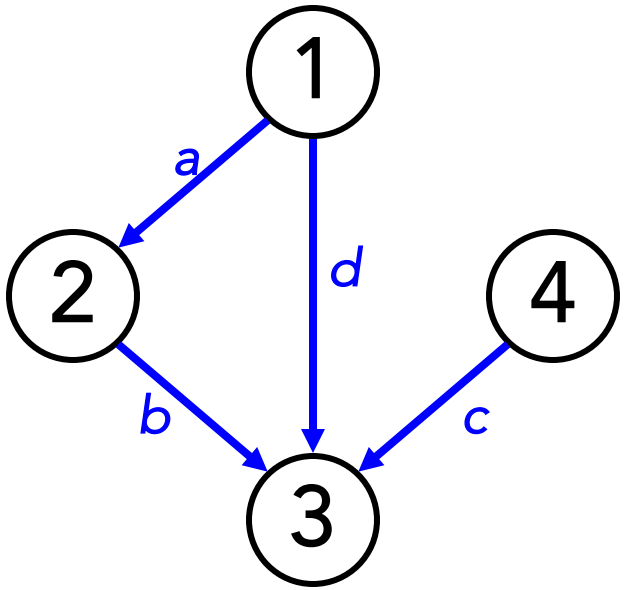
Graphs: Adjacency Matrix

- The adjacency matrix is a dense representation
  - Usually too much storage for large graphs
  - But can be very efficient for small graphs

- Most large interesting graphs are sparse
  - E.g., planar graphs, in which no edges cross, have $|E| = O(|V|)$ by Euler's formula
  - *adjacency list* - more appropriate representation

Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to v
- Example:
  - $adj[1] = \{2,3\}$
  - $adj[2] = \{3\}$
  - $adj[3] = \{\}$
  - $adj[4] = \{3\}$
- Variation: can also keep a list of edges coming *into* vertex



Single Source Shortest Paths

- Applications
  - Internet packet routing, Flight reservations, Driving directions, …

- Let G be a weighted graph. The length of a path P is the sum of the weights of the edges of P. If P consists of $e_0, e_1, \ldots, e_{k-1}$ then length of P, denoted as $w(P)$, is defined as

$$w(P) = \sum_{i=0}^{k-1} w(e_i)$$

- The distance of a vertex $\boldsymbol{v}$ from a vertex $\boldsymbol{s}$ is the length of a shortest path between $\boldsymbol{s}$ and v. denoted as $\boldsymbol{d(s, v)}$.

  $\boldsymbol{d(s, v)} = +\infty$ if no path exists

Single Source Shortest Paths

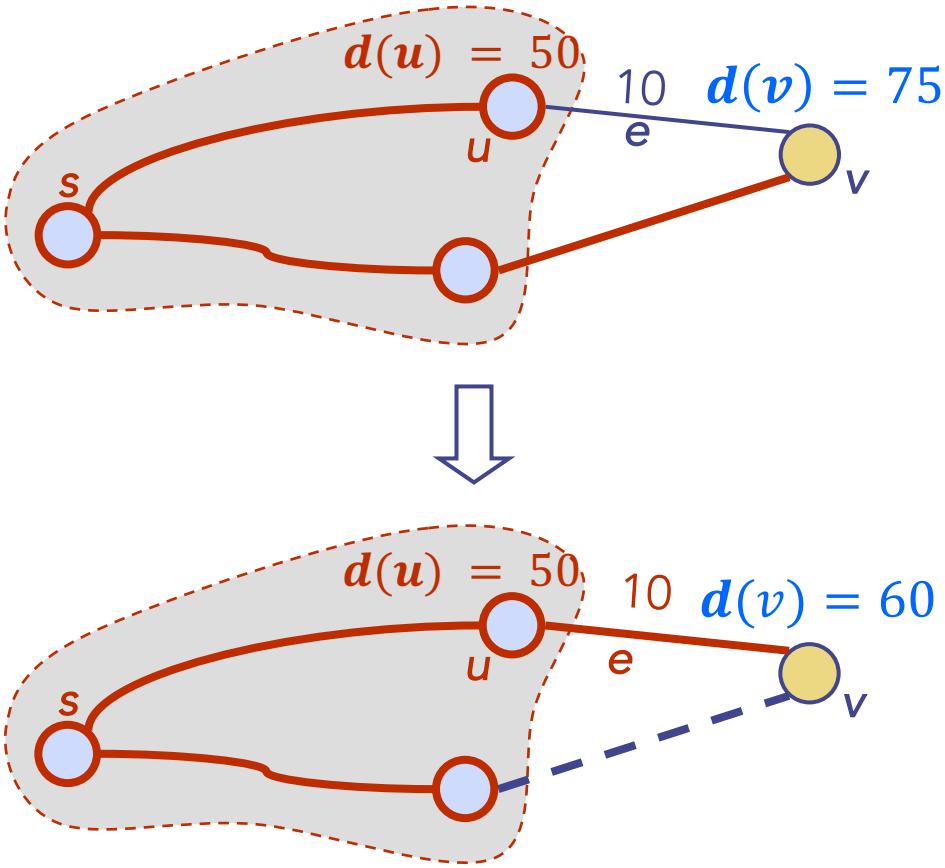
- Dijkstra's algorithm:
  - It computes the shortest distances of all the vertices from a given start vertex s
- Assumptions:
  - graph is connected
  - edges are directed/undirected
  - edge weights are nonnegative, i.e. $w(e) \geq 0$

Dijkstra's Algorithm

- Greedy method
- We grow a "cloud or set" of vertices, beginning with start vertex s and eventually covering all the vertices
- For each vertex v a label $d(v)$: 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

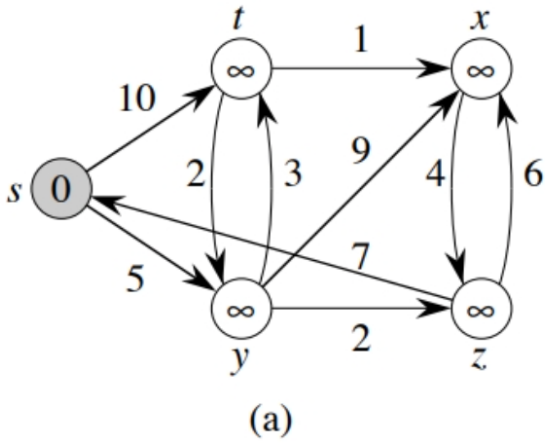
Edge Relaxation

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

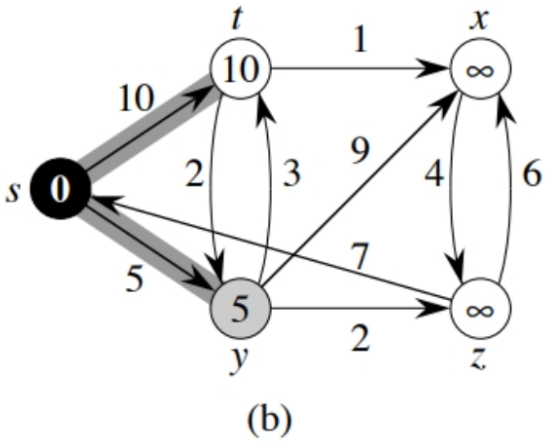



```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```

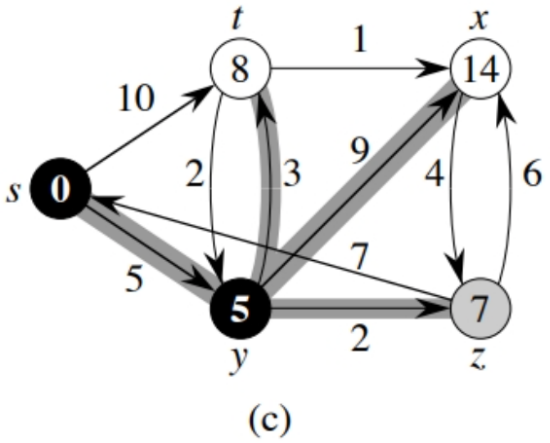
Example 1



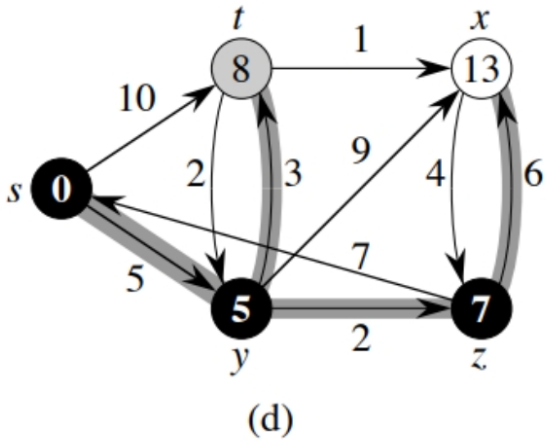
(a)



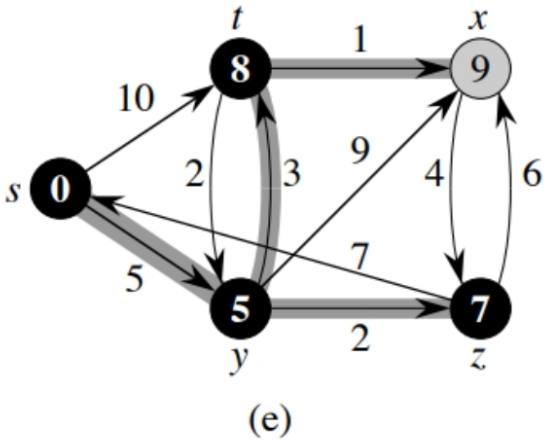
(b)



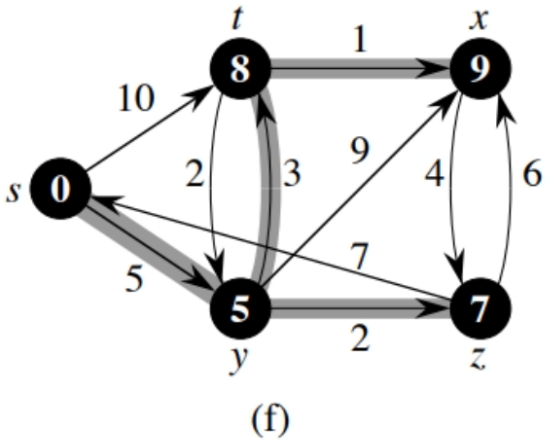
(c)



(d)

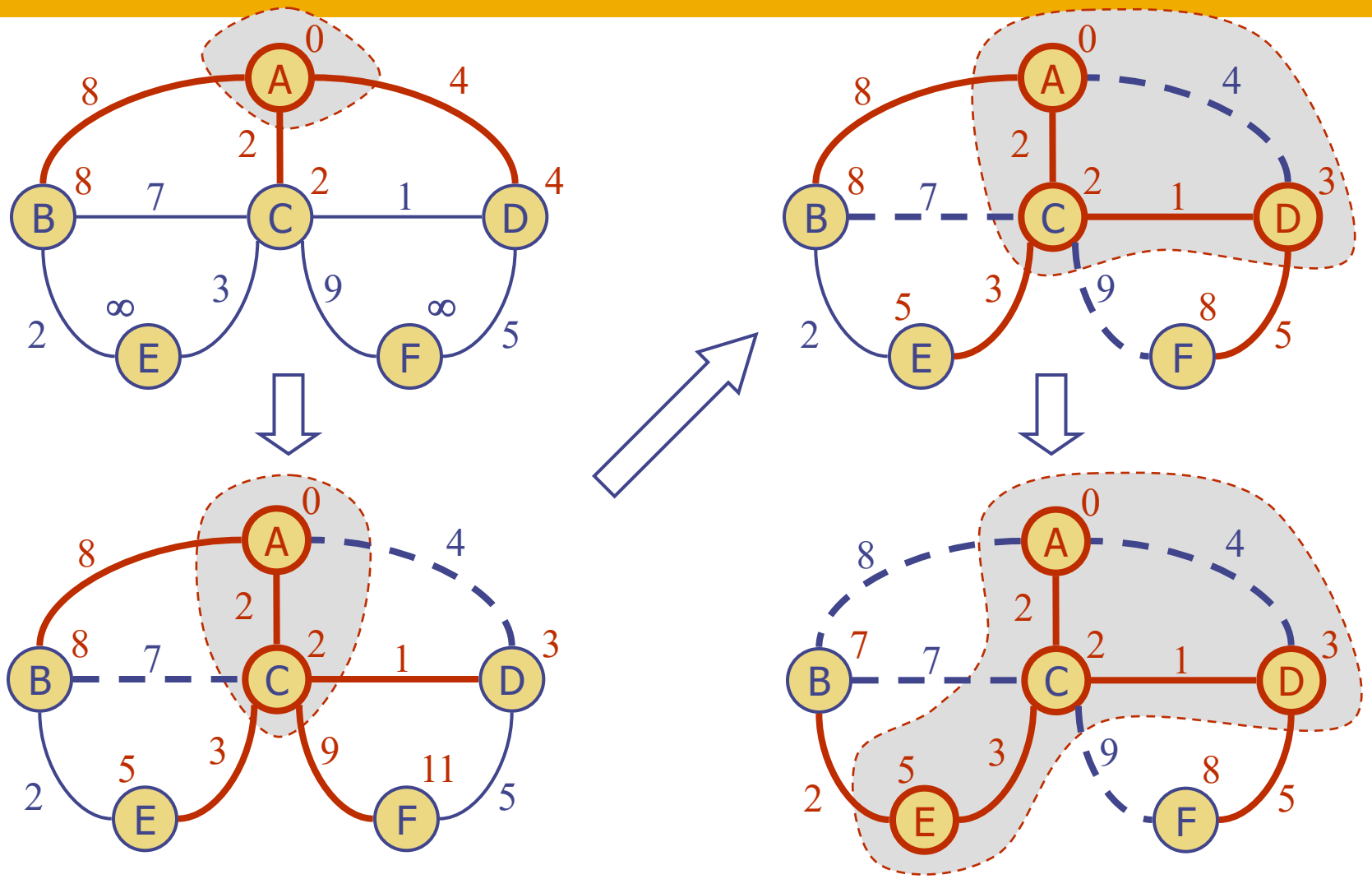


(e)

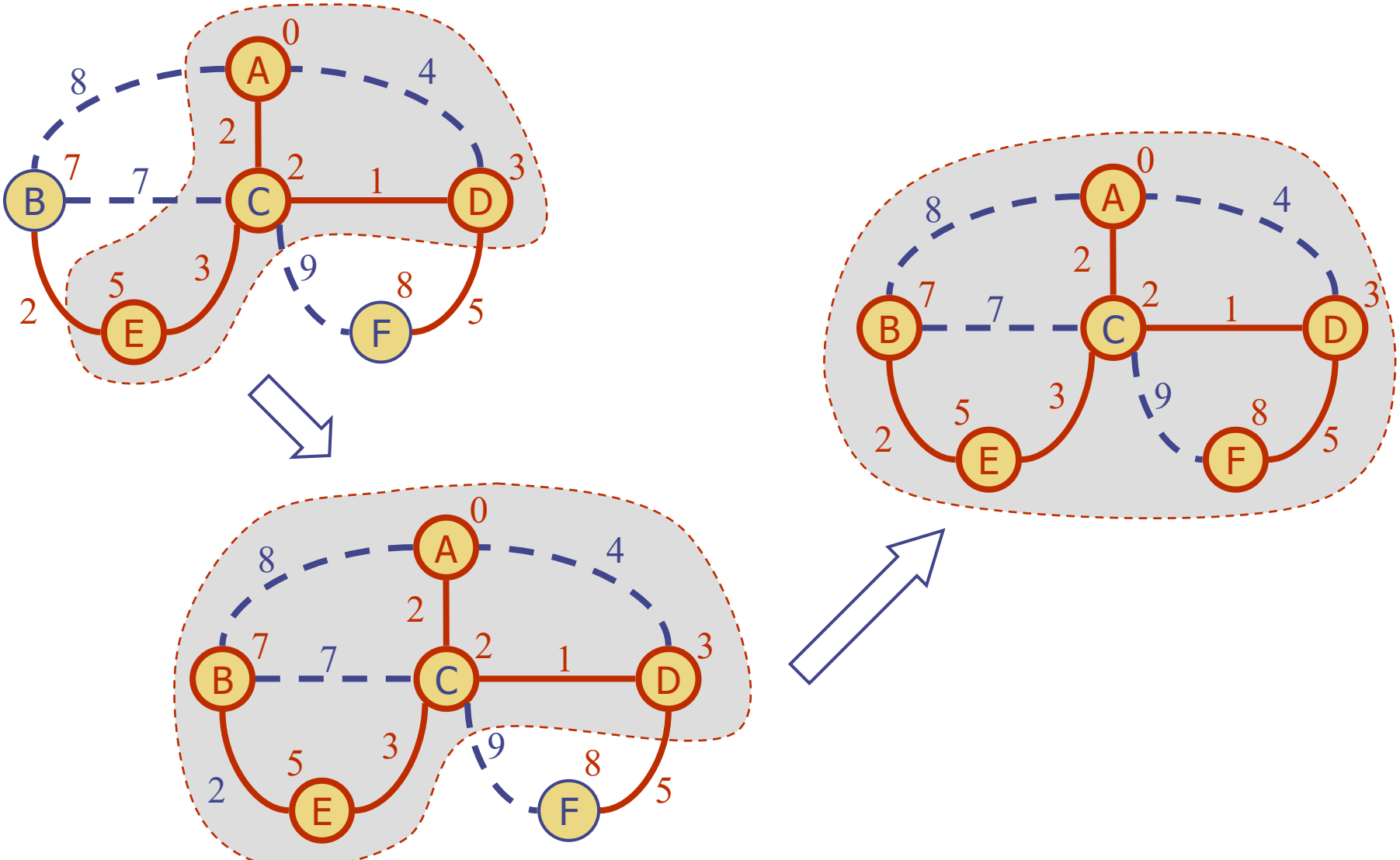


(f)

Example 2


A
B
C
D
E
F
0
8
4
2
8
7
2
1
4
∞
3
9
∞
2
5
8
4
2
8
7
2
1
3
5
3
9
8
2
5
3
0
8
4
2
8
7
2
1
3
5
3
9
11
2
5
0
8
4
2
7
7
2
1
3
5
3
9
8
2
5

Example 2 (cont.)



Dijkstra's Algorithm and Runtime

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

What will be the total running time?
$O(m \log n)$ using binary heap for Q

ExtractMin() - How many times called?

decraseKey() - How many times called?

Dijkstra's Algorithm and Runtime

- The time to build the binary min-heap is $O(n \log n)$.
- Each EXTRACT-MIN operation then takes time $O(\log n)$.
- Each DECREASE-KEY (i.e. Relax) operation takes time $O(\log n)$. and there are still at most $O(m)$ such operations.
- The total running time is therefore $O(n \log n + m \log n)$ or $O((n+m) \log n)$, which is $O(m \log n)$ if all vertices are reachable from the source.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Shortest-path Tree

- We can extend Dijkstra's algorithm to return a tree of shortest paths from the start vertex to all other vertices

- In the edge relaxation step - update the parent label

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```

Correctness of the Algorithm

- Dijkstra's algorithm is a greedy algorithm
  - Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in V - S to add to set S , we say that it uses a greedy strategy.

- The following theorem and its corollary show, Dijkstra's algorithm does indeed compute shortest paths.

- Correct because maintains following two properties:
  - for every known vertex (inside the cloud or within the set S), recorded distance is the shortest distance to that vertex from the source vertex
  - for every unknown vertex v, its recorded distance is shortest path distance to v from source vertex, considering only currently known vertices and v

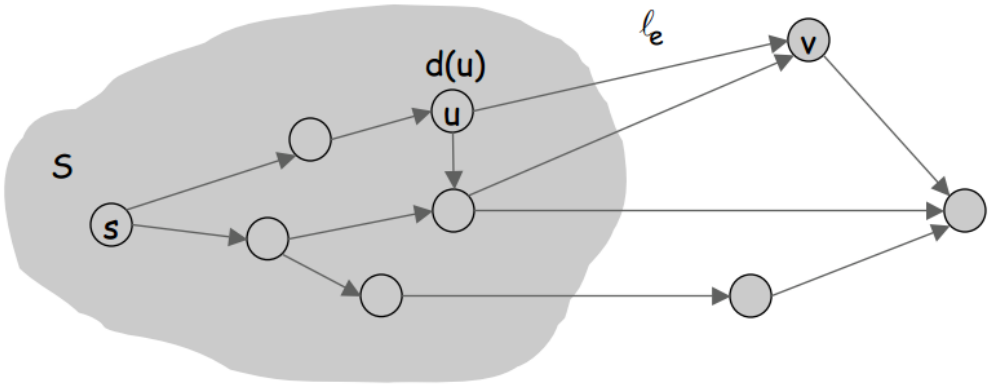
Dijkstra's Algorithm: Proof of Correctness

- Dijkstra's algorithm:
  - Maintain a set of explored nodes S for which we have determined the shortest path distance $d(u)$ from s to u.
  - Initialize $S = \{s\}$, $d(s) = 0$.
  - Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e$$

shortest path to some u (i.e. known vertex), followed by a single edge (u, v)

  - add v to S, and set $d(v) = \pi(v)$.

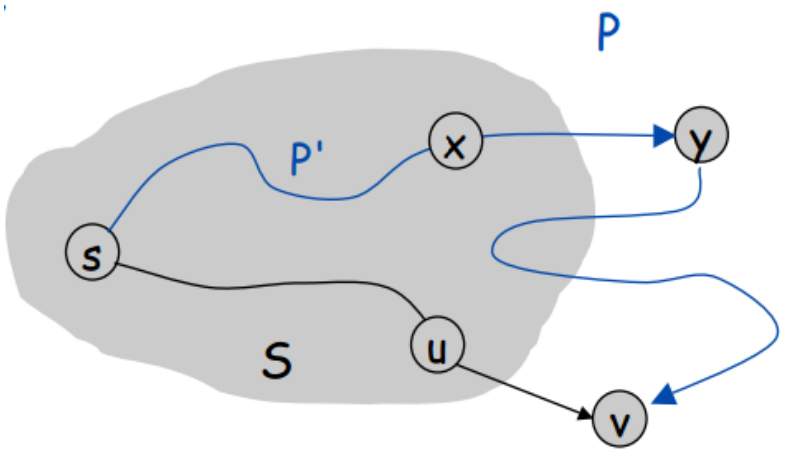


Dijkstra's Algorithm: Proof of Correctness

- Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest $s \sim u$ path.
  - Proof. (by induction on |S|)
- Base case: $|S| = 1$ is trivial.
- Inductive hypothesis: Assume true for $|S| = k \geq 1$

Dijkstra's Algorithm: Proof of Correctness

- Let v be next node added to S, and let $\mathbf{e}(u, v)$ be the chosen edge.
- The shortest $s \sim u$ path plus $\mathbf{e}(u, v)$ is an $s \sim v$ path of length $\pi(v)$.
- Consider any $s \sim v$ path P. We'll observe that it's no shorter than $\pi(v)$.
- Let $\mathrm{e}(x, y)$ be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



$$\pi(v) = d(u) + l(u, v)$$

$$\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq \pi(y) \geq \pi(v)$$

nonnegative weights

inductive hypothesis

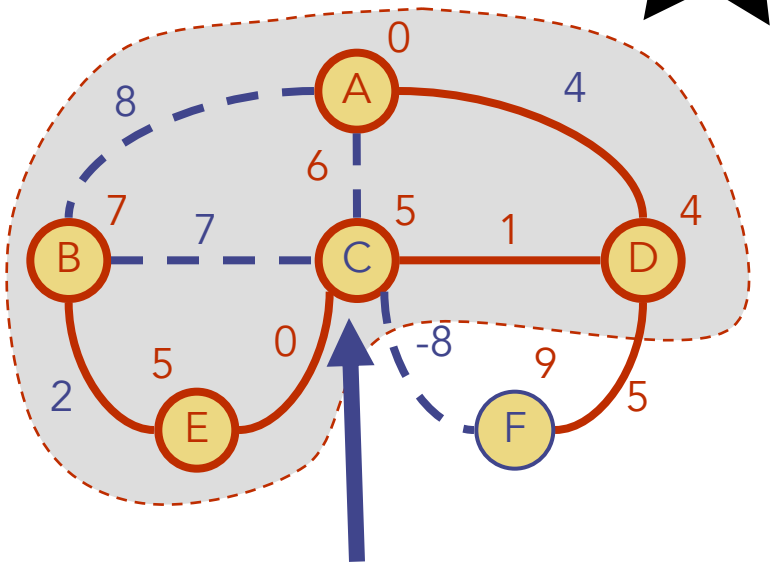
defn of $\pi(y)$

Dijkstra chose v instead of y

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 $d(C) = 5$!

Shortest Path Properties

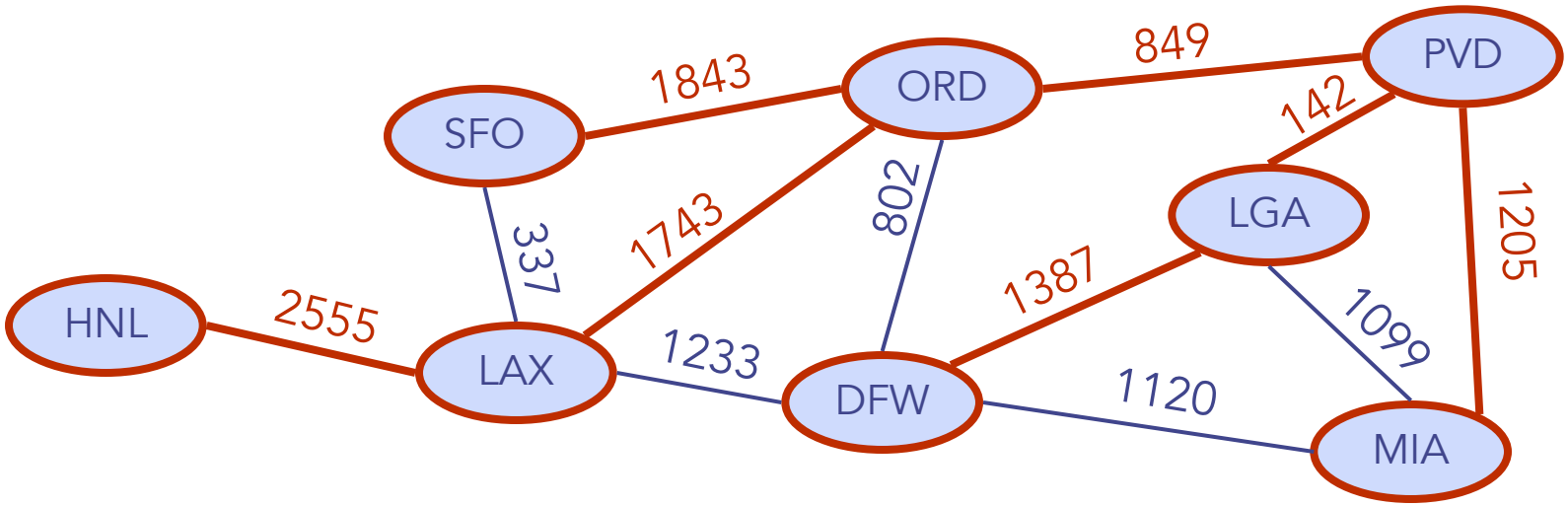
Property 1:

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



References

- Algorithm Design: Foundations, Analysis, and Internet Examples. Michael T. Goodrich and Roberto Tamassia. John Wiley & Sons.
- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

Thank you!