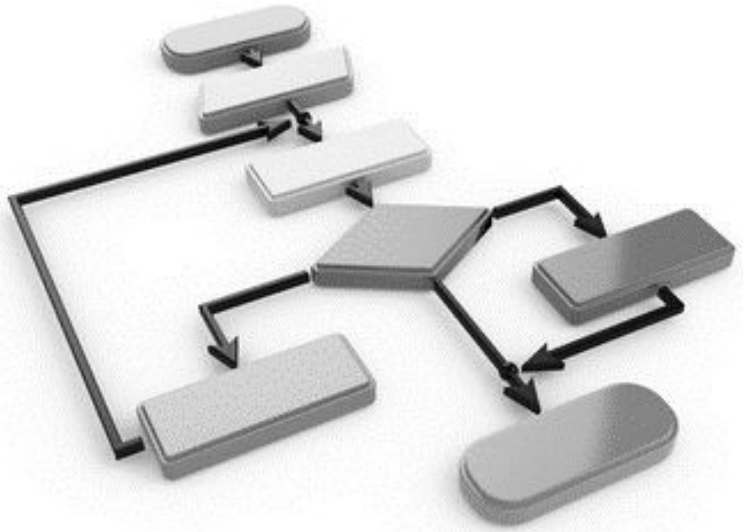


Graph-Based Algorithms

CSE 301: Combinatorial Optimization

Shortest Path Problems

Modeling problems as graph problems:

Road map is a weighted graph:

vertices = cities

edges = road segments between cities

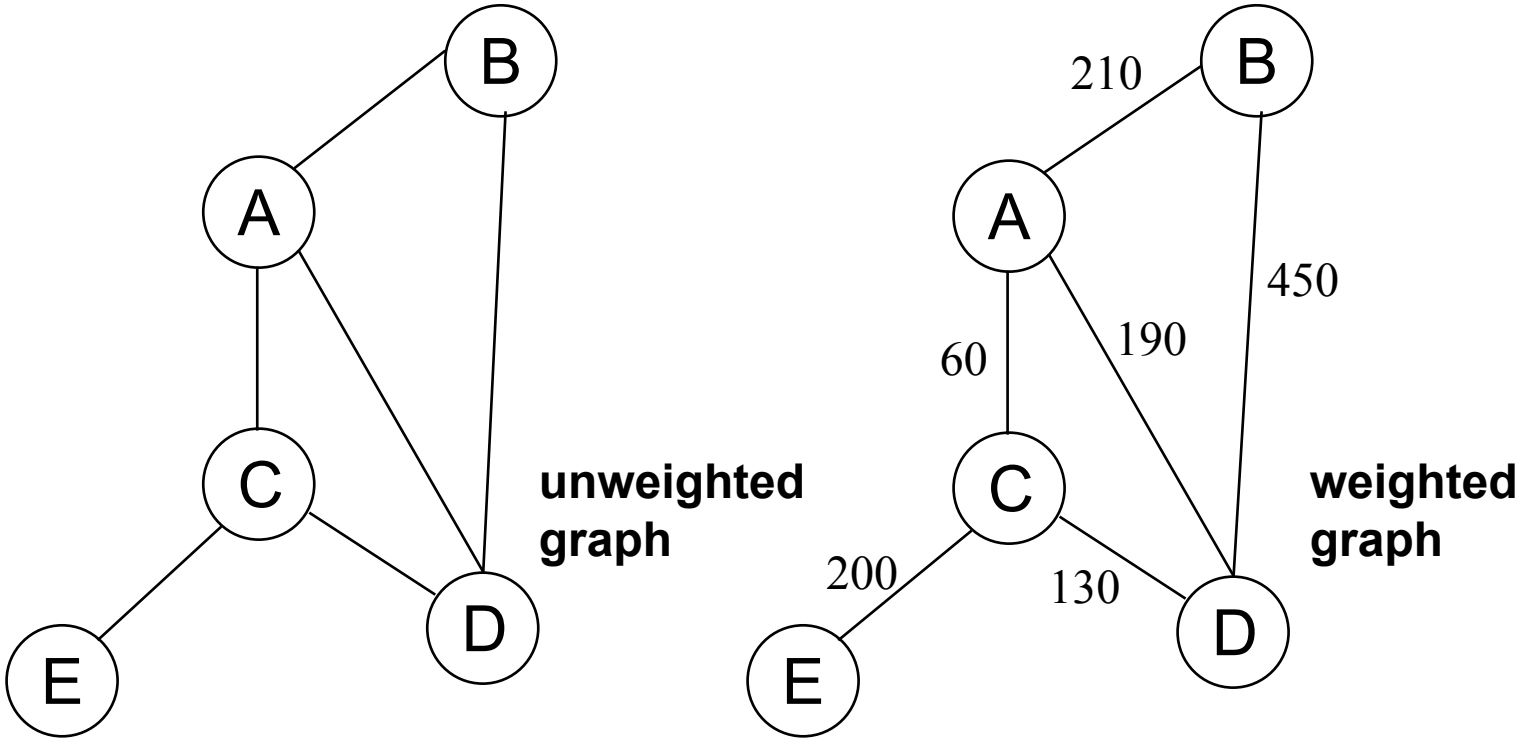
edge weights = road distances

Goal: find a shortest path between two vertices (cities)

Shortest Path Problems

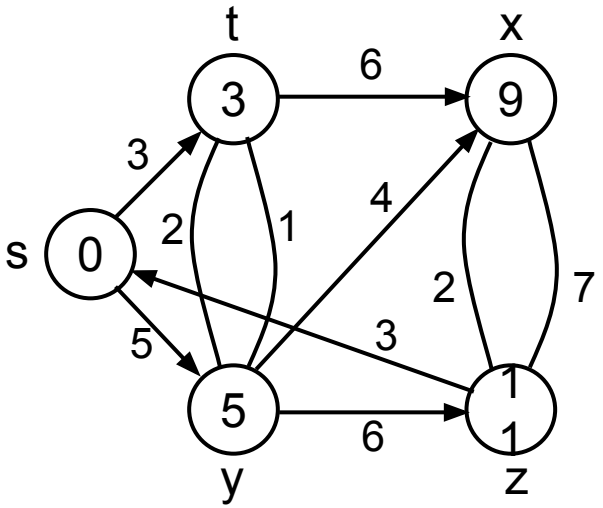
What is shortest path ?

- shortest length between two vertices for an unweighted graph:
- smallest **cost** between two vertices *for* a weighted graph:



Shortest Path Problems

- **Input:**
  - Directed graph G = (V, E)
  - Weight function w : E → **R**



- **Weight of path** p = $\langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- **Shortest-path weight** from u to v:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there exists a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Variants of Shortest Paths

Single-source shortest path

Given G = (V, E), find a shortest path from a given source vertex s to each vertex $v \in V$

Single-destination shortest path

Find a shortest path to a given destination vertex **t** from each vertex **v**

Reverse the direction of each edge $\Rightarrow$ single-source

Single-pair shortest path

Find a shortest path from **u** to **v** for given vertices **u** and **v**

Solve the single-source problem

All-pairs shortest-paths

Find a shortest path from **u** to **v** for every pair of vertices **u** and **v**

Optimal Substructure of Shortest Paths

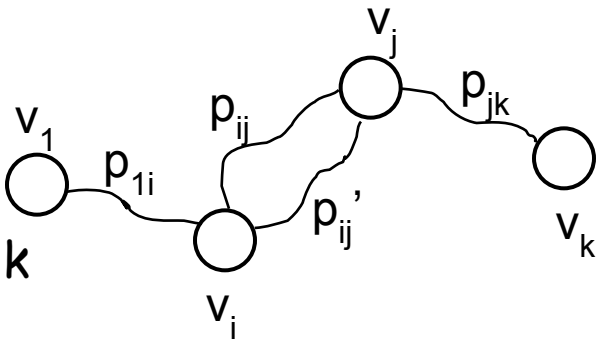
Given:

A weighted, directed graph G = (V, E)

A weight function w: E → **R**,

A shortest path $p = \langle v_1, v_2, \ldots, v_k \rangle$ from v_1 to v_k

A subpath of p: $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$, with $1 \le i \le j \le k$



Then: p_{ij} is a shortest path from v_i to v_j

Proof: Let $p = v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$

$\Rightarrow w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$

Assume $\exists\ p_{ij}'$ from v_i to v_j with $w(p_{ij}') < w(p_{ij})$

Adding $w(p_{1i}) + w(p_{jk})$ in both sides of this inequality:

$\Rightarrow w(p') = w(p_{1i}) + w(p_{ij}') + w(p_{jk}) < w(p_{1i}) + w(p_{ij}) + w(p_{jk}) = w(p)$

So there is a path *p'* from v_1 to v_k which is shorter than the shortest path *p* between them; but this contradicts our initial assumption that *p* is the shortest path.

Shortest-Path Idea

- Recall: $\delta(u,v)$ ≡ total weight/cost of the shortest path from u to v.
- All SSSP algorithms maintain a field d[u] for every vertex u. d[u] will be an estimate of $\delta(s,u)$. As the algorithm progresses, we will refine d[u] until, at termination,

$$d[u] = \delta(s,u).$$

- Whenever we discover a new shortest path to u, we update d[u].

  In fact, d[u] will always be an *overestimate* of $\delta(s,u)$:

$$d[u] \geq \delta(s,u)$$

- We'll use $\pi[u]$ to point to the parent (or predecessor) of u on the shortest path from s to u. We update $\pi[u]$ when we update d[u].
- At the end, π will induce a tree, called **shortest path tree**.

Initialization

Alg.: INITIALIZE-SINGLE-SOURCE(V, s)

1. **for** each $v \in V$

2. **do** $d[v] \leftarrow \infty$

3. $\pi[v] \leftarrow$ NIL

4. $d[s] \leftarrow 0$

All the shortest-paths algorithms start with INITIALIZE-SINGLE-SOURCE

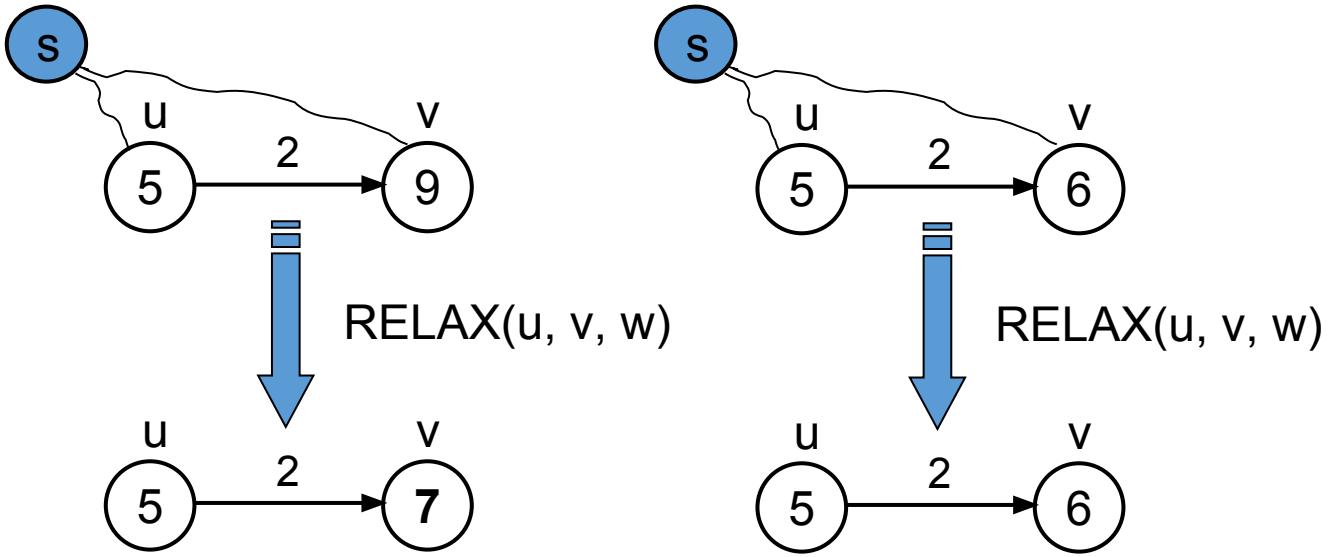
Relaxation

Relaxing an edge (u, v) = testing whether we can improve the shortest path to v found so far by going through u

If $d[v] > d[u] + w(u, v)$

we can improve the shortest path to v

$\Rightarrow$ update $d[v]$ and $\pi[v]$



After relaxation: $d[v] \leq d[u] + w(u, v)$

RELAX(u, v, w)

1. **if** $d[v] > d[u] + w(u, v)$
2. **then** $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$

All the single-source shortest-paths algorithms

- start by calling INIT-SINGLE-SOURCE
- then relax edges

The algorithms differ in the order and how many times they relax each edge

Dijkstra's Algorithm

(pronounced "DIKE-stra")

Solves Single-source Shortest Path (SSSP) problem:

When there is no negative-weight edges: $w(u, v) > 0 \ \forall (u, v) \in E$

Maintains two sets of vertices:

K = vertices whose final shortest-path weights have already been determined

Q = vertices in V – K (Q is a min-priority queue)

Keys in Q are estimates of shortest-path weights (d[v])

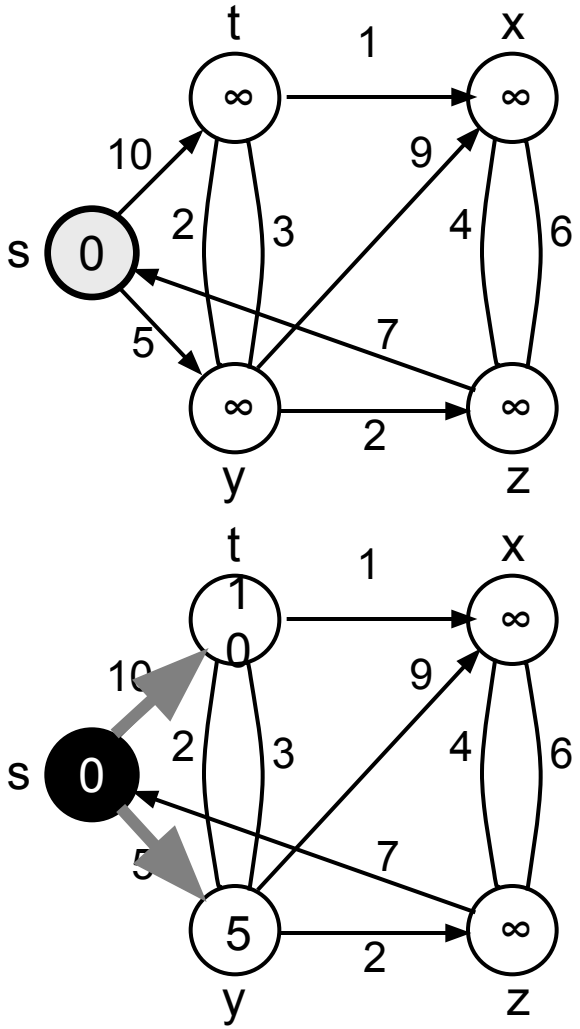
Repeatedly select a vertex $u \in V - K$, with the minimum shortest-path estimate d[v] and RELAXs its incident edges.

Dijkstra (G, w, s)

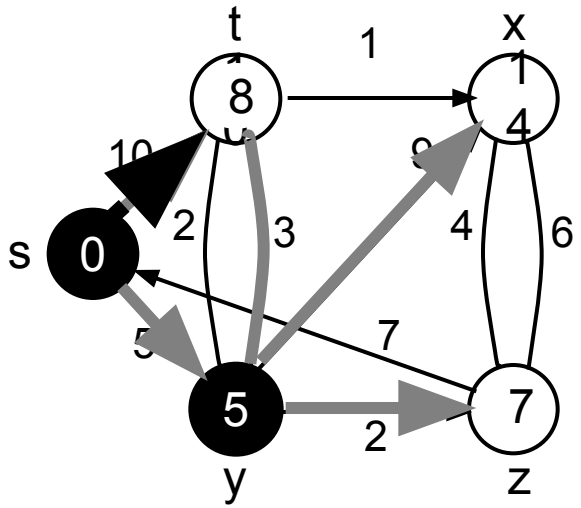
1. INITIALIZE-SINGLE-SOURCE(V, s)
2. Q ← V[G]
3. **while** Q ≠ ∅
4. **do** u ← EXTRACT-MIN(Q)

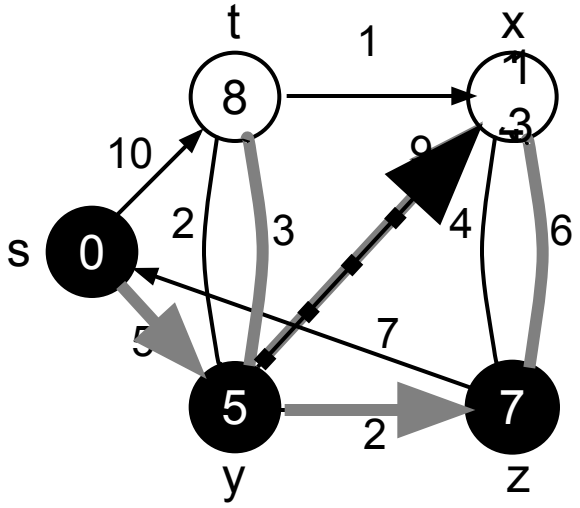
   for each vertex v ∈ Adj[u]

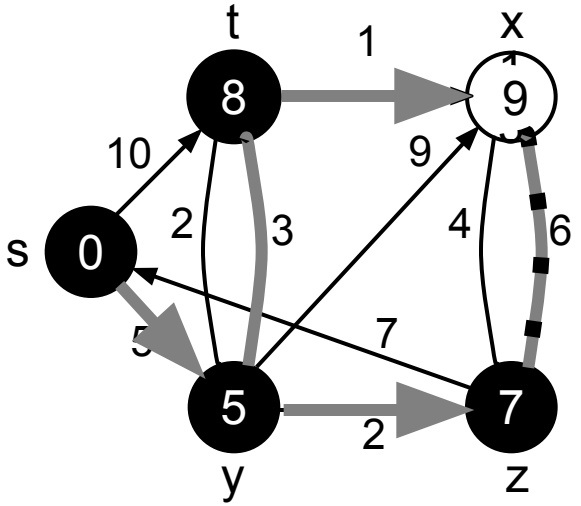
1. **do** RELAX(u, v, w)

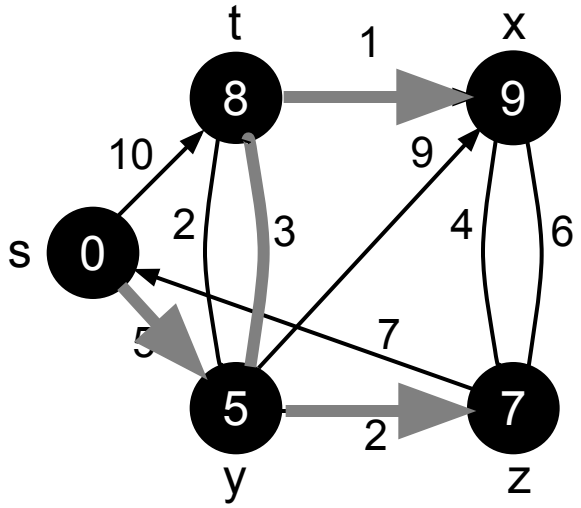


Example

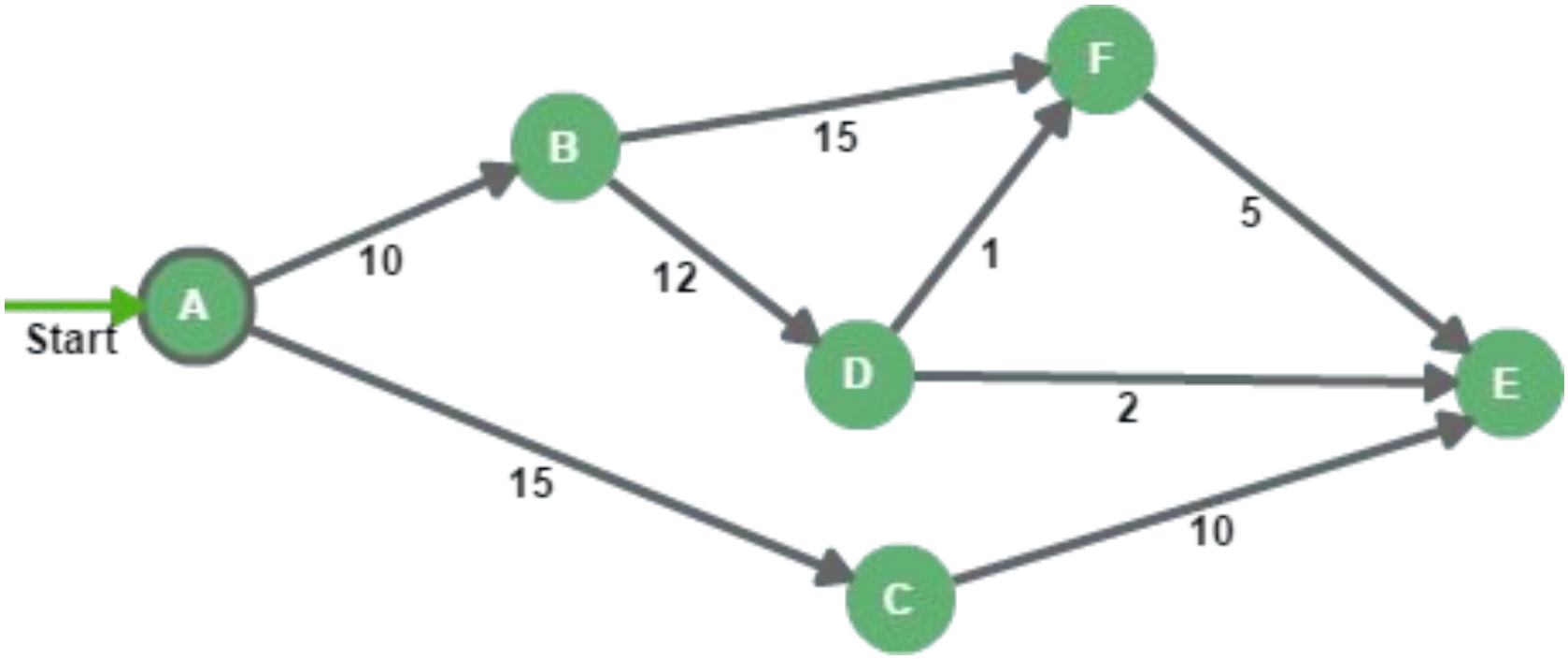

t
x
1
8
14
10
9
s
0
2
3
4
6
5
7
7
2
y
z


t
x
1
8
13
10
9
s
0
2
3
4
6
5
7
7
2
y
z


t
x
1
8
9
10
9
s
0
2
3
4
6
5
7
7
2
y
z


t
x
1
8
9
10
9
s
0
2
3
4
6
5
7
7
2
y
z

Practice



Dijkstra (G, w, s) – Time Complexity

1. INITIALIZE-SINGLE-SOURCE(V, s) ← $\Theta(V)$
2. K ← ∅
3. Q ← V[G] ← $\Theta(V)$ time to build min-heap
4. **while** Q ≠ ∅ ← Executed $\Theta(V)$ times
5. **do** u ← EXTRACT-MIN(Q) ← V times, $\Theta(T_E)$ each time
6. K ← K ∪ {u}
7. **for** each vertex v ∈ Adj[u]
8. **do** RELAX(u, v, w) ← $\Theta(E)$ times, $\Theta(T_D)$ each time

Total running time: $\Theta(V + V \times T_E + E \times T_D)$

T_E = time taken by an EXTRACT-MIN operation
T_D = time taken by a DECREASE-KEY operation

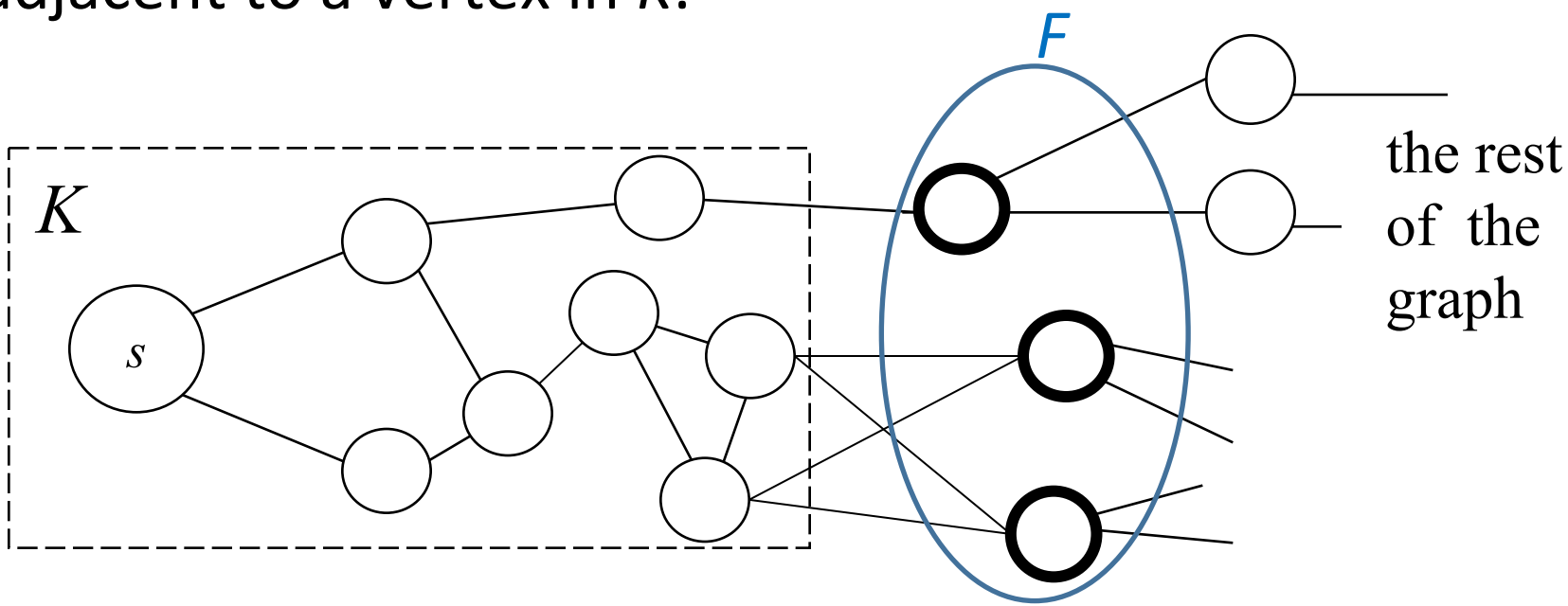
Dijkstra's Time complexity (cont.)

1. Priority queue is an array.
   EXTRACT-MIN in $\Theta(V)$ time, DECREASE-KEY in $\Theta(1)$
   Total time: $\Theta(V + VV + E) = \Theta(V^2)$
2. ("Modified Dijkstra")
   Priority queue is a binary (standard) heap.
   EXTRACT-MIN and DECREASE-KEY takes $\Theta(\lg V)$ time each
   Total time: $\Theta(V \lg V + E \lg V) = \Theta(E \lg V)$
3. Priority queue is Fibonacci heap. (Of theoretical interest only.)
   EXTRACT-MIN in $\Theta(\lg V)$,
   DECREASE-KEY in $\Theta(1)$ (amortized)
   Total time: $\Theta(V \lg V + E)$

Why Does Dijkstra's Algorithm Work?

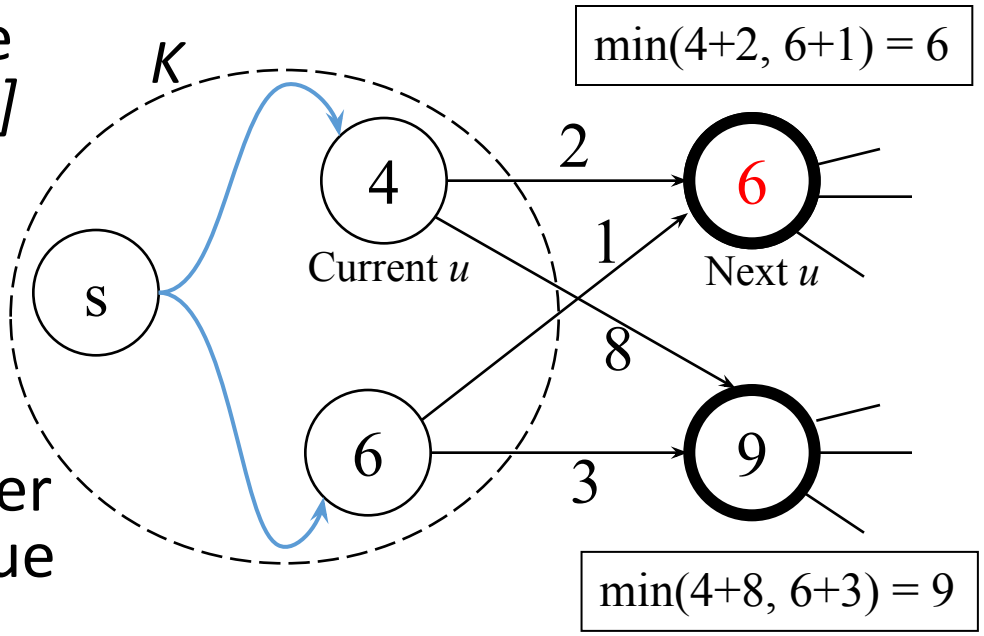
- It works when all edge weights are ≥ 0.
- Why?
- It maintains a set *K* containing all vertices whose shortest paths from the source vertex *s* are known (i.e. $d[u] = \delta(s,u)$ for all *u* in *K*).

  Now look at the "frontier" *F* of *K* — *i.e.,* all vertices adjacent to a vertex in *K*.



Dijkstra's: Theorem

After the end of each iteration of while loop in Dijkstra's algorithm the following is true for each frontier vertex *u*: *d[u]* is the weight of the shortest path to *u* going through only vertices in *K*.



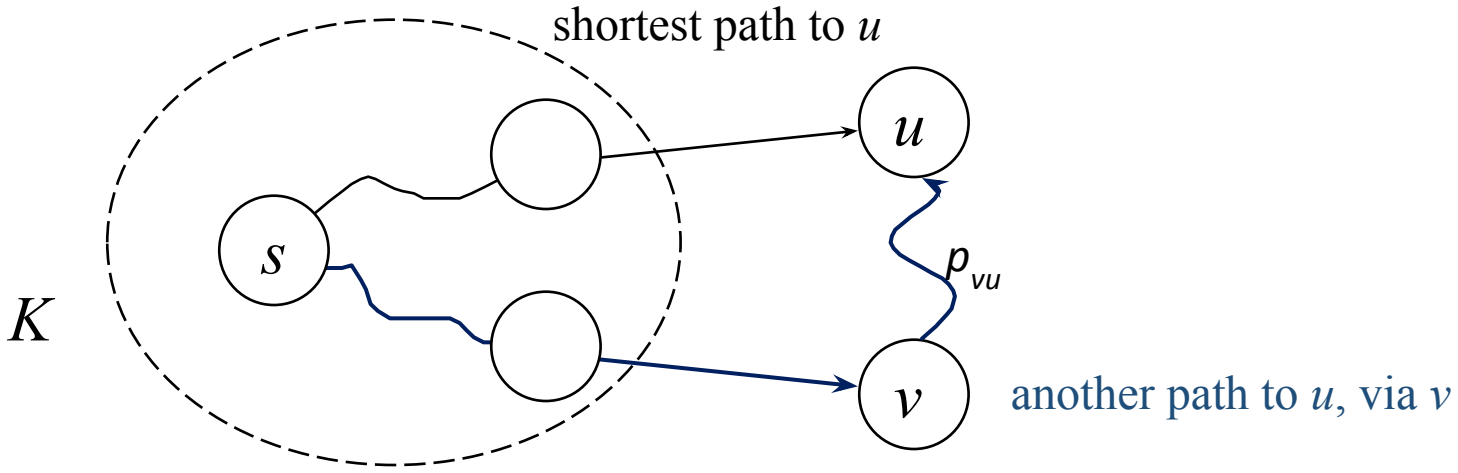
The algorithm picks the frontier vertex *u* with the smallest value of *d[u]* in the next iteration.

Claim: If u is chosen (via the Extract-Min opertaion in line 5) in the beginning of an iteration then ***d[u] = δ(s,u)***

Dijkstra's: Proof

By construction, d[*u*] is the weight of the shortest path to *u* going through only vertices in *K*.

If there exist another path *p* to *u* that contains some vertices not in *K,* then that path must leave *K*, go to a node *v* on the frontier and then reach *u* from there (via a sub-path p_{vu}).



Dijkstra's: Proof (Contd.)

The weight of this path, $w(p) = d[v]+w(p_{vu}) \geq d[v]$, since $w(p_{vu}) \geq 0$ (because edge weights are non-negative).

But $d[v] \geq d[u]$, otherwise u wouldn't be chosen in line 5.

$\therefore w(p) \geq d[u]$. So there is no path to u which is shorter than $d[u]$, *i.e.*, $d[u]$ is the shortest path distance of u from s, *i.e.*,

$$d[u] = \delta(s,u)$$