طراحی الگوریتم ها

جلسه ۱۴ و ۱۵

ملکی مجد

مباحث

مساله کوتاه ترین مسیرها از یک مبدا تا سایر راس ها

preliminaries
Relaxation technique
Bellman Ford algorithm
Shortest path in DAG
Dijkstra’s algorithm

مبحث Single-Source Shortest Path از فصل ۲۴ کتاب CLRS تدریس می شود.

مسئله کوتاه ترین مسیر

- کوتاهترین مسیر از دانشکده کامپیوتر تا در اصلی دانشگاه کدام است؟
  - (از ساختمان یا فضای سبز رد نمی شیم)
  - یک نقشه از دانشگاه داریم که فاصله های هر دو تقاطعی بر روی آن مشخص شده است.

- یک راه ممکن:
  - **همه مسیرها** را در نظر بگیریم و طول مسیرها را با توجه به قسمت های تشکیل دهنده آن حساب کنیم
  - با توجه به طول های بدست آمده، مسیر با کوتاهترین طول را پیدا کنیم
  - مشکلات این راه چیست؟

حل مساله کوتاه ترین مسیر

- یک سری مسیرها ارزش بررسی ندارند
  - مثلا از در استخر خارج بشیم و از خارج دانشگاه به در اصلی برسیم
    - (مسیرهایی که مشخصا طول بیشتری را طی می کنیم)
  - دور بوفه یک دور بزنیم
    - (گذشتن از دورها)

- چه مسیرهایی را بررسی کنیم؟

- اگر مساله پیچیده تر شود و تعداد مسیرها زیاد باشد چه کنیم؟

- برنامه های کاربردی مثل waze و google map طول کوتاهترین مسیر را (البته با توجه به داده های برخط) برای ما حساب می کنند ☺
  - مشابه داخلی: Neshan

تمرکز بحث

- در این جلسه حل مسئله کوتاهترین فاصله از مبدا خاص تا همه مقصدها را در نظر می گیریم.

الگوریتم کارایی برای محاسبه طول کوتاهترین مسیرها را یاد می گیریم

می توانیم مسئله مسیر یابی را با یک گراف مدل کنیم

تقاطع ها راس های گراف هستند

مسیر بین هر دو تقاطع یک یال گراف هست که وزن این یال فاصله دو تقاطع است

هدف پیدا کردن کوتاه ترین مسیر بین دو راس این گراف است

shortest-paths problem in graph

-formal definition

- Given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow \boldsymbol{R}$ mapping edges to real-valued weights.

- The ***weight*** of path $p = < v_0, v_1, \ldots, v_k >$ is the sum of the weights of its constituent edges:

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

δ pronounced delta

- ***shortest-path weight*** from u to v is $\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightarrow} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$

- A ***shortest path*** from vertex u to vertex v is defined as any path p with weight $w(p) = \delta(u, v)$.

Minimize other metrics

- Edge weights can represent metrics other than distances,
  - Cost, time, penalty, loss
  - Any quantity that accommodates linearly along a path

Shortest-paths in unweighted graphs

- The **breadth-first-search (BFS) algorithm** from Section 22.2 is a **shortest-paths algorithm** that works on **unweighted** graphs, that is, graphs in which each edge can be considered to have unit weight.

single-source shortest-paths problem

- Given a graph $G = (V, E)$, we want to find a shortest path from a given ***source*** vertex $s \in V$ to each vertex $v \in V$.

There are **other** variants can be solved with SSSP algorithm

See them in the following slides.

Variant (1)

- **Single-destination shortest-paths problem:**

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

By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

Variant (2)

- **Single-pair shortest-path problem:**

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

If we solve the single-source problem with source vertex u, we solve this problem also.

➢noted, **no** algorithms for this problem are known that run asymptotically faster than the best single-source algorithms in the worst case.

Variant (3)

- **All-pairs shortest-paths problem:**

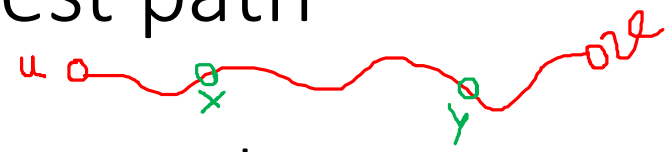
Find a shortest path from u to v for every pair of vertices u and v.

It can be solved by running a single source algorithm once from each vertex (however, it can usually **be solved faster)**.

Additionally, its structure is of interest in its own right.

- Later (chapter 25 of CLRS), we addresses the all-pairs problem in detail.

Optimal substructure of a shortest path



- Shortest-paths algorithms typically rely on the property that a **shortest path between two vertices contains other shortest paths within it**.

- **greedy method**
  - Dijkstra's algorithm (to solve the single-source shortest-paths problem on a weighted, directed graph in which all edge weights are **nonnegative**)
- **dynamic-programming**
  - Floyd-Warshall algorithm (to find shortest paths between all pairs of vertices)

Lemma 24.1
(*Subpaths of shortest paths are shortest paths*)

- Given a weighted, directed graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with weight function $\boldsymbol{w}: \boldsymbol{E} \rightarrow \boldsymbol{R}$

let $p = < v_0, v_2, \ldots, v_k >$ be a shortest path from vertex v_1 to vertex v_k and,

let $p_{ij} = < v_i, v_{i+1}, \ldots, v_j >$ be the subpath of p from vertex v_i to vertex v_j .

for any i and j such that $0 \leq i \leq j \leq k$,

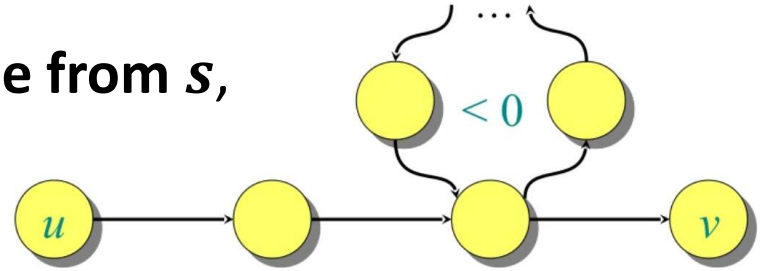
Then, p_{ij} is a shortest path from v_i to v_j .

Proof : contradiction!

Proof If we decompose path p into $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, then we have that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k. ■

Negative-weight edges

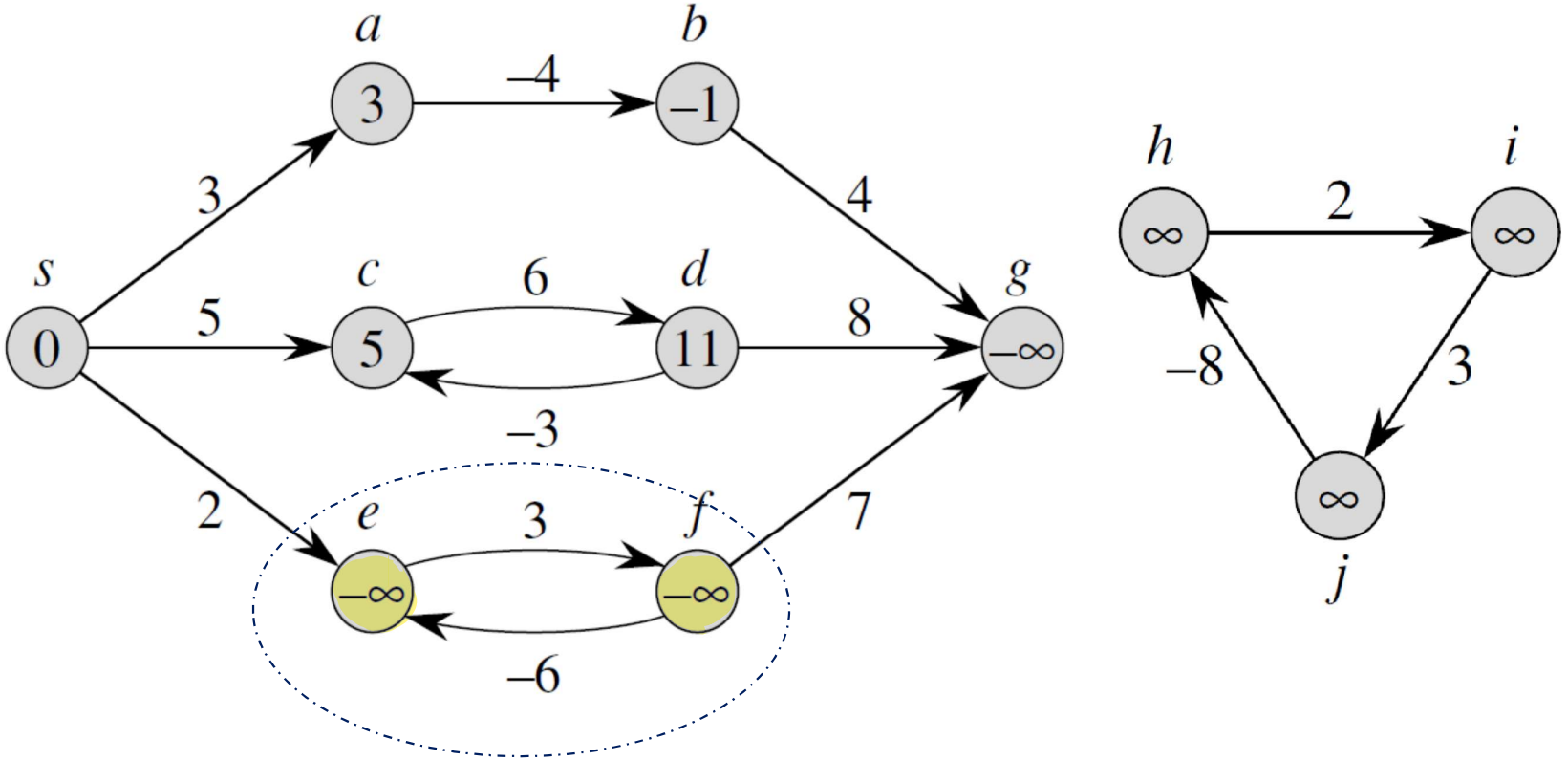
- If there is a **negative-weight cycle reachable from *s***,
  shortest-path weights are **not well defined**.



- If there is a negative-weight cycle on some path from s to v,
  we define $\delta(s, v) = -\infty$.

- If the graph $G = (V, E)$ contains no negative weight cycles reachable from the source s, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains **well defined**,

understand the effect of negative weight edges



Cycles

- Can a shortest path contain a cycle?

NO

- As a result:

  we can restrict our attention to shortest paths of **at most $|V| - 1$ edges**.

Representing shortest paths

- Wish to compute not only shortest-path weights,

  but the **vertices on shortest paths** as well

So

- Given a graph $G = (V, E)$, we maintain for each vertex $v \in V$

  a ***predecessor*** $\pi[v]$ that is either another vertex or NIL.

- π attributes:

the chain of predecessors originating at a vertex v runs backwards along a shortest path from s to v

Remember PRINT-PATH(G, s, v).

Predecessor subgraph G_π

- ***predecessor subgraph*** $G_\pi = (V_\pi, E_\pi)$ induced by the π values.

$$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}.$$

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$$

The algorithms in this chapter(24) have the property that

at termination G_π is a "shortest-paths tree"

- During the execution of a shortest-paths algorithm, however, the π values need not indicate shortest paths

Shortest-paths tree

- A ***shortest-paths tree*** rooted at s is

  a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that

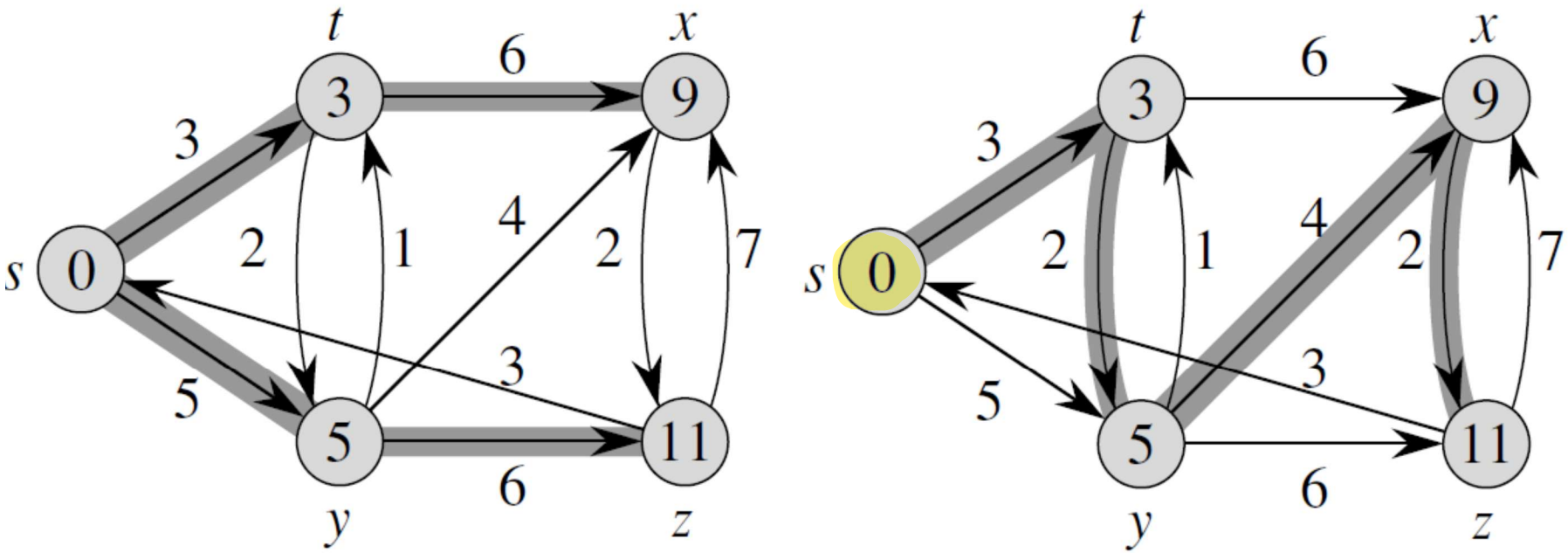
  I. V' is the set of vertices reachable from s in G,

  II. G' forms a rooted tree with root s, and

  III. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Shortest paths are not necessarily unique, and neither are shortest-paths trees.

two shortest paths trees with the same root

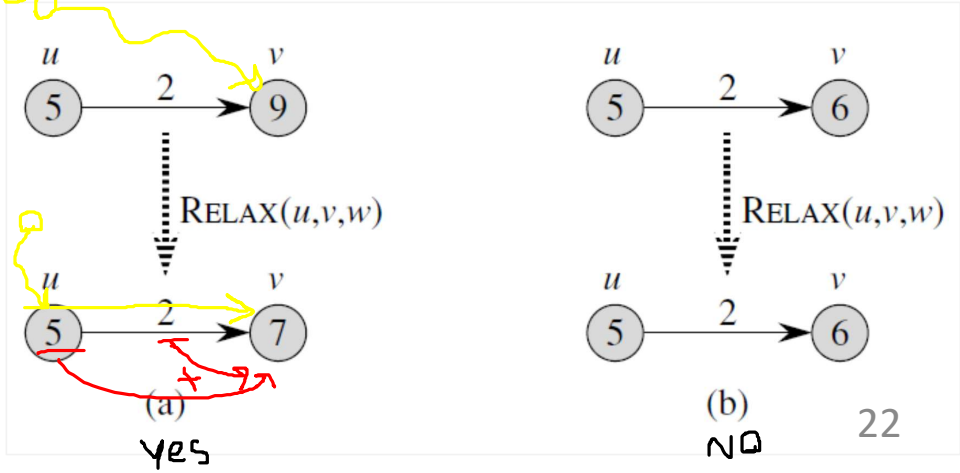


Technique of *relaxation*

The algorithms in this chapter use this technique

- $d[v]$ a ***shortest-path estimate*** *(during the algorithm execution)*
  - is an upper bound on the weight of a shortest path from source s to v

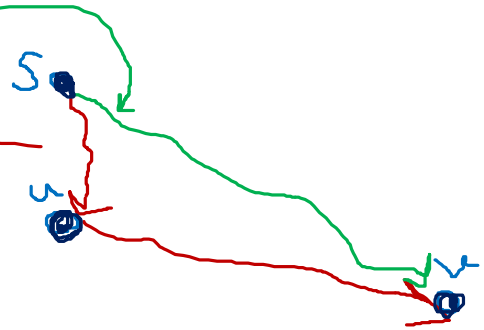
- The process of ***relaxing*** an edge *(u, v)* consists of testing whether we can improve the shortest path to *v* found so far by going through *u* and, if so, updating *d*[*v*] and π[*v*].



Each algorithm in this chapter calls INITIALIZE-SINGLE-SOURCE and then repeatedly relaxes edges.

INITIALIZE-SINGLE-SOURCE(G, s)
1 **for** each vertex $v \in V[G]$
2 **do** $d[v] \leftarrow \infty$
3 $\pi[v] \leftarrow NIL$
4 $d[s] \leftarrow 0$

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



- Relaxation is a means by which shortest path estimates and predecessors change.

- The algorithms in this chapter differ in how many times they relax each edge and the order in which they relax edges.
  - In Dijkstra's algorithm and the shortest-paths algorithm for directed acyclic graphs, each edge is relaxed exactly once.
  - In the Bellman-Ford algorithm, each edge is relaxed many times.

Properties of shortest paths and relaxation

- **Triangle inequality** (Lemma 24.10)

  For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

- **Upper-bound property** (Lemma 24.11)

  We always have $d[v] \geq \delta(s, v)$ for all vertices $v \in V$, and once $d[v]$ achieves the value *δ(s, v)*, it never changes.

- **No-path property** (Corollary 24.12)

  If there is no path from *s* to *v*, then we always have $d[v] = \delta(s, v) = \infty$.

- **Convergence property** (Lemma 24.14)

  If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $d[v] = \delta(s, v)$ always afterward.

Continue…

- **Path-relaxation property** (Lemma 24.15)

  If $p = < V_0, V_1, \ldots, V_k >$ is a shortest path from s $(= V_0)$ to V_k , and the edges of p are relaxed in the order $(V_0, V_1), (V_1, V_2), \ldots, (V_{k-1}, V_k)$, then $d[V_k] = \delta(s, V_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

- **Predecessor-subgraph property** (Lemma 24.17)

  Once $d[v] = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s. (implicitly assume that the graph is initialized with a call to INITIALIZESINGLE-SOURCE(G, s) and that the only way that shortest-path estimates and the predecessor subgraph change are by some sequence of relaxation steps.)

Algorithms in the following.

- Bellman-Ford algorithm
  - Solves the single-source shortest-paths problem in the general case in which edges can have negative weight.
  - Remarkable in its **simplicity**
  - Detecting whether a negative-weight cycle is reachable from the source

- A linear-time algorithm
  - Computing shortest paths from a single source in a **directed acyclic graph**

- Dijkstra's algorithm
  - a lower running time than the Bellman-Ford algorithm
    requires the edge weights to be nonnegative

All algorithms in this chapter(24) assume

- Directed graph *G* is stored in the adjacency-list representation.
- Stored with each edge is its weight
- Traverse each adjacency list, we can determine the edge weights in *O(*1*)* time per edge.

The Bellman-Ford algorithm

- Given a weighted, directed graph $G = (V, E)$ with source *s* and weight function $w : E \rightarrow \boldsymbol{R}$, (allow negative-weight edges)

algorithm

- Returns a **Boolean value** indicating whether there is a **negative-weight cycle** that is reachable from the source
- If there is no such cycle, the **algorithm produces the shortest paths and their weights**.

The Bellman-Ford algorithm

- After initializing the d and π values of all vertices,

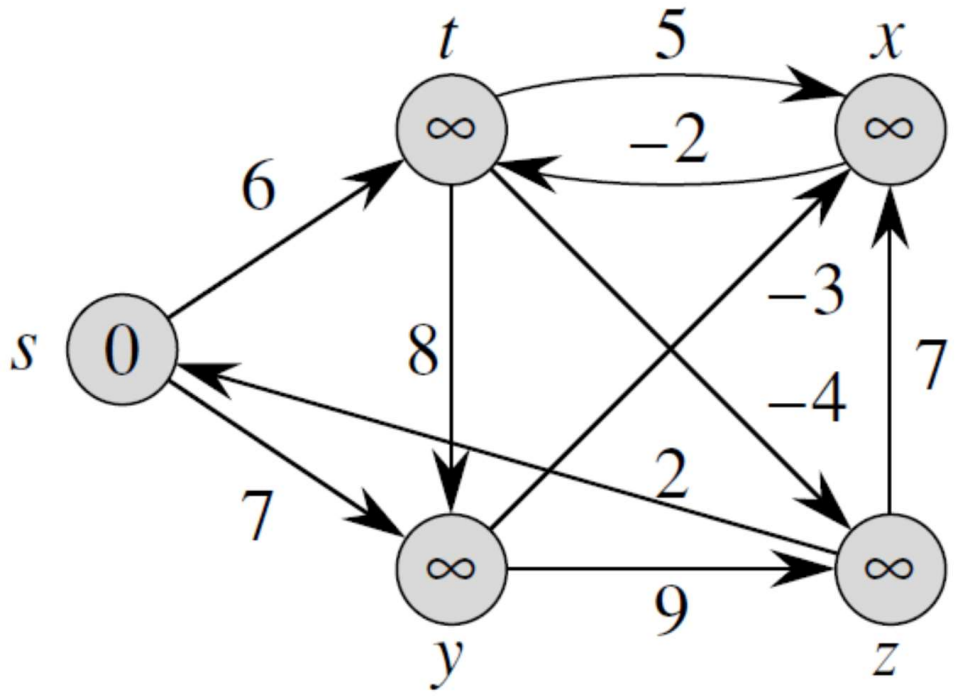
the algorithm makes $|\boldsymbol{V}| - \mathbf{1}$ **passes over the edges of the graph**

the algorithm uses **relaxation**, progressively decreasing an estimate $d[v]$ on the weight of a shortest path from the source s to each vertex $v \in V$ **until** it achieves the actual shortest-path weight $\delta(s, v)$.

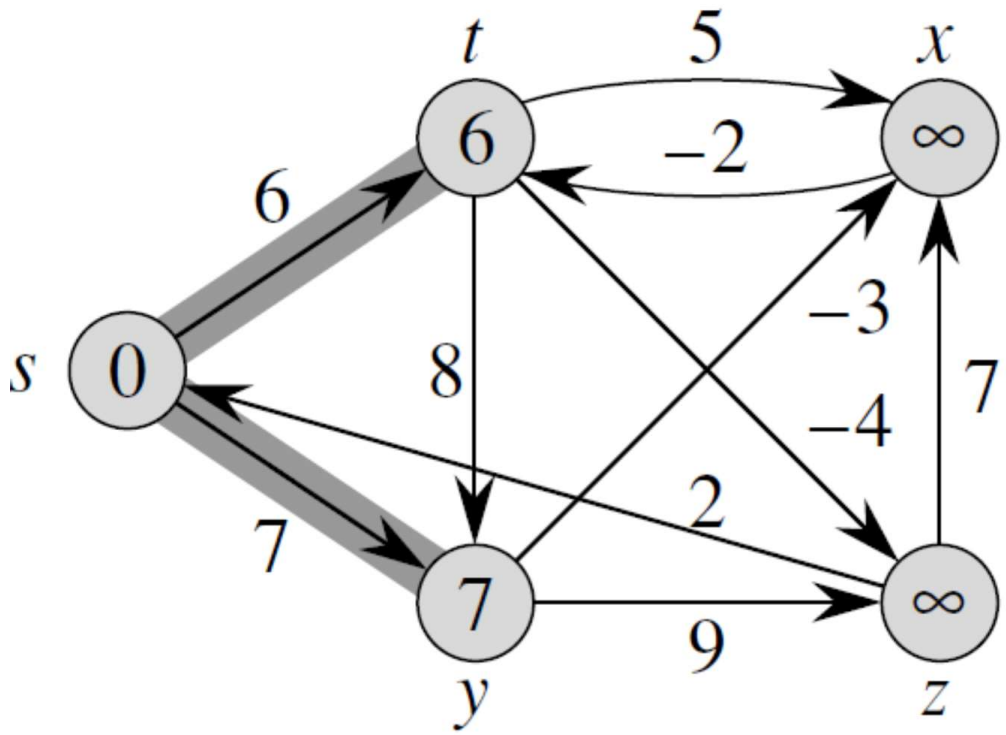
The Bellman-Ford algorithm

BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 **for** $i \leftarrow 1$ **to** $|V[G]| - 1$
3 **do for** each edge $(u, v) \in E[G]$
4 **do** RELAX(u, v, w)
5 **for** each edge $(u, v) \in E[G]$
6 **do if** $d[v] > d[u] + w(u, v)$
7 **then return** FALSE
8 **return** TRUE

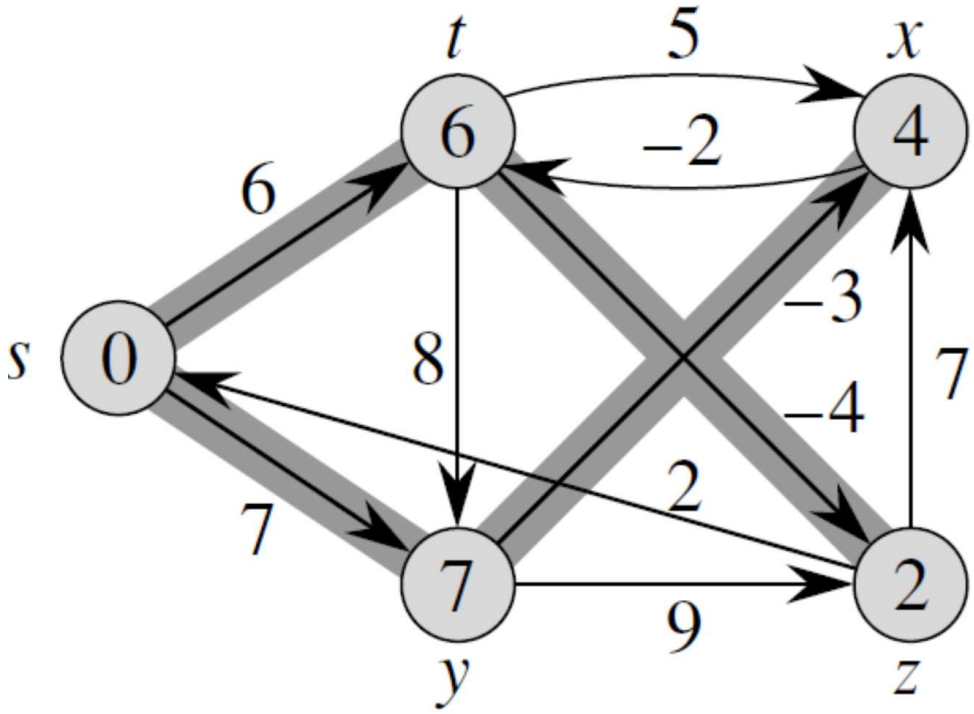
Example of the Bellman-Ford algorithm



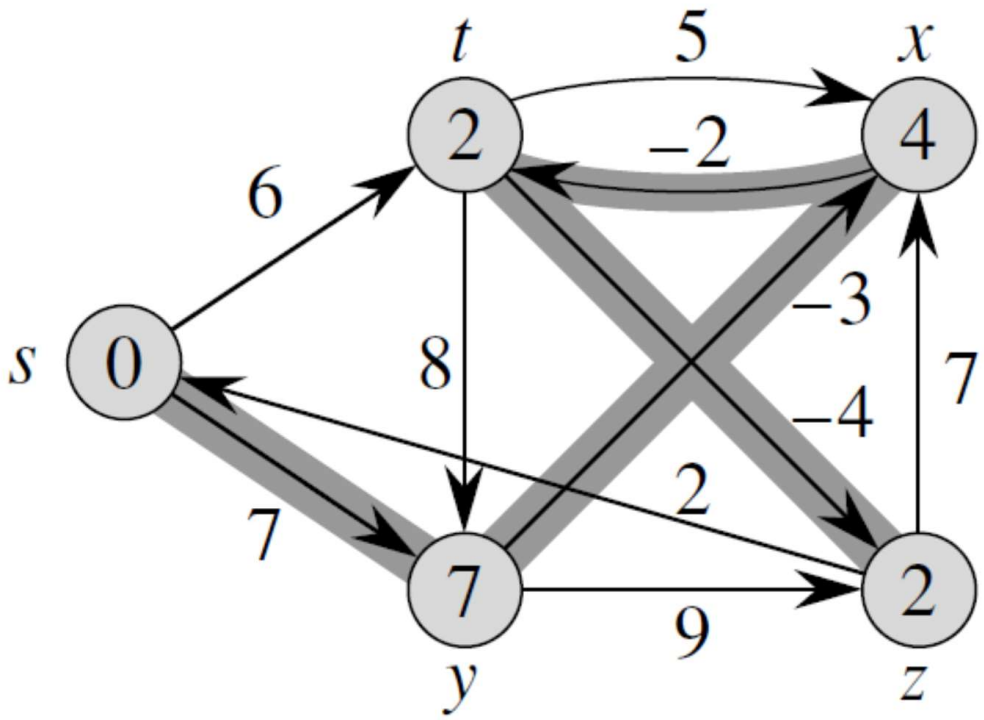
Pass 1



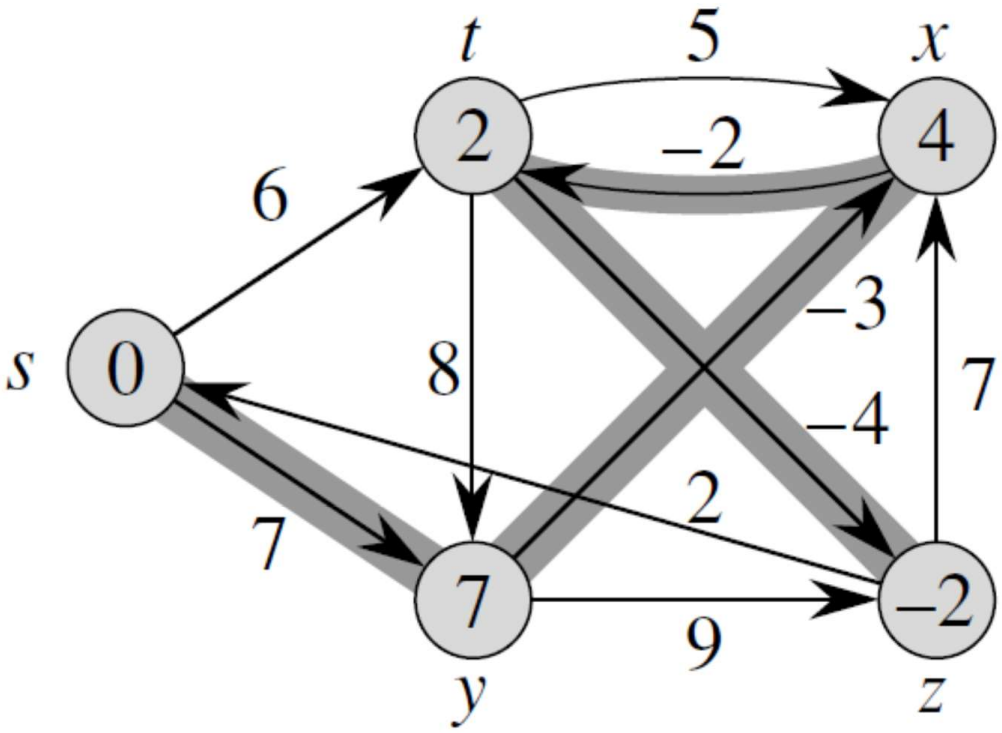
Pass 2



Pass 3



Pass 4 (V-1)

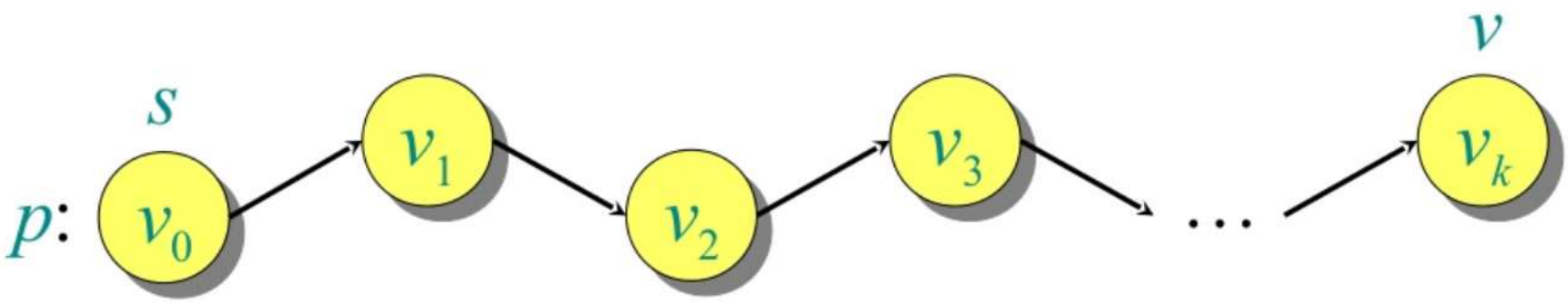


Time complexity

```
BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i ← 1 to |V[G]| − 1
3         do for each edge (u, v) ∈ E[G]
4                 do RELAX(u, v, w)
5 for each edge (u, v) ∈ E[G]
6         do if d[v] > d[u] + w(u, v)
7                 then return FALSE
8 return TRUE
```

The Bellman-Ford algorithm runs in time *O(V E)*,

Idea of
Correctness of the Bellman-Ford algorithm



Shortest paths in DAG

Single-source shortest paths in directed acyclic graphs

- By relaxing the edges of a weighted dag (directed acyclic graph) $G = (V, E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\boldsymbol{\Theta(V + E)}$ time.

- Shortest paths are always well defined in a dag, since even if there are negative-weight edges, **no negative-weight cycles** can exist.
- If there is a path from vertex u to vertex v then u precedes v in the topological sort.

DAG-SHORTEST-PATHS$(\boldsymbol{G}, \boldsymbol{w}, \boldsymbol{s})$

1 Topologically sort the vertices of G

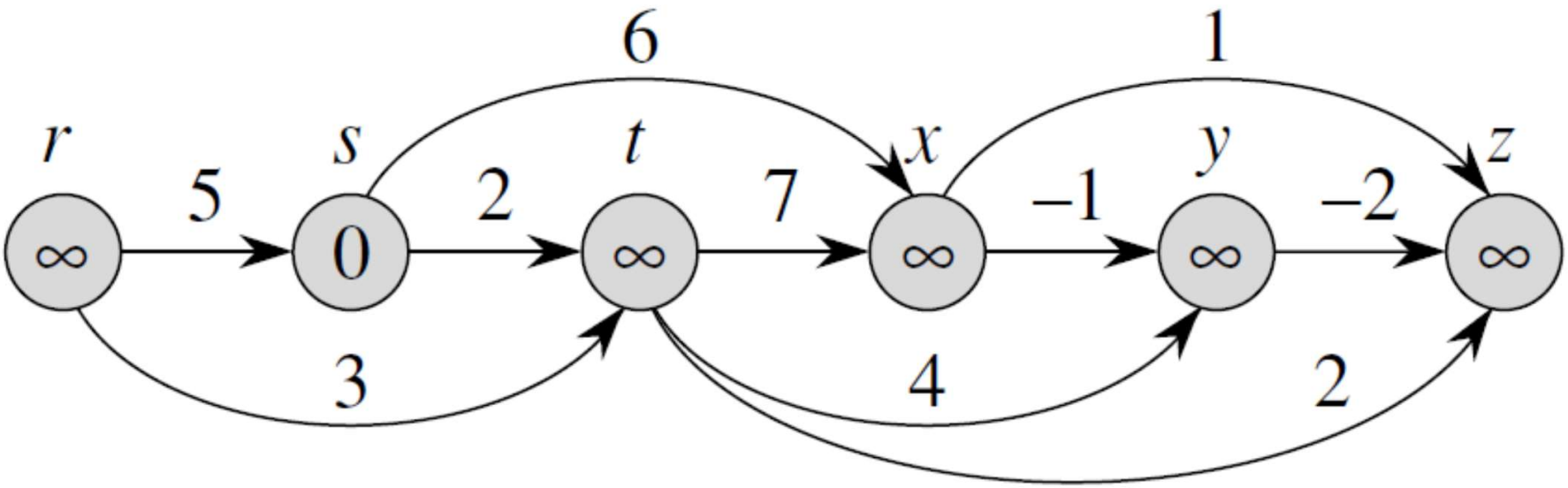
2 INITIALIZE-SINGLE-SOURCE(G, s)

3 **for** each vertex u, taken in topologically sorted order

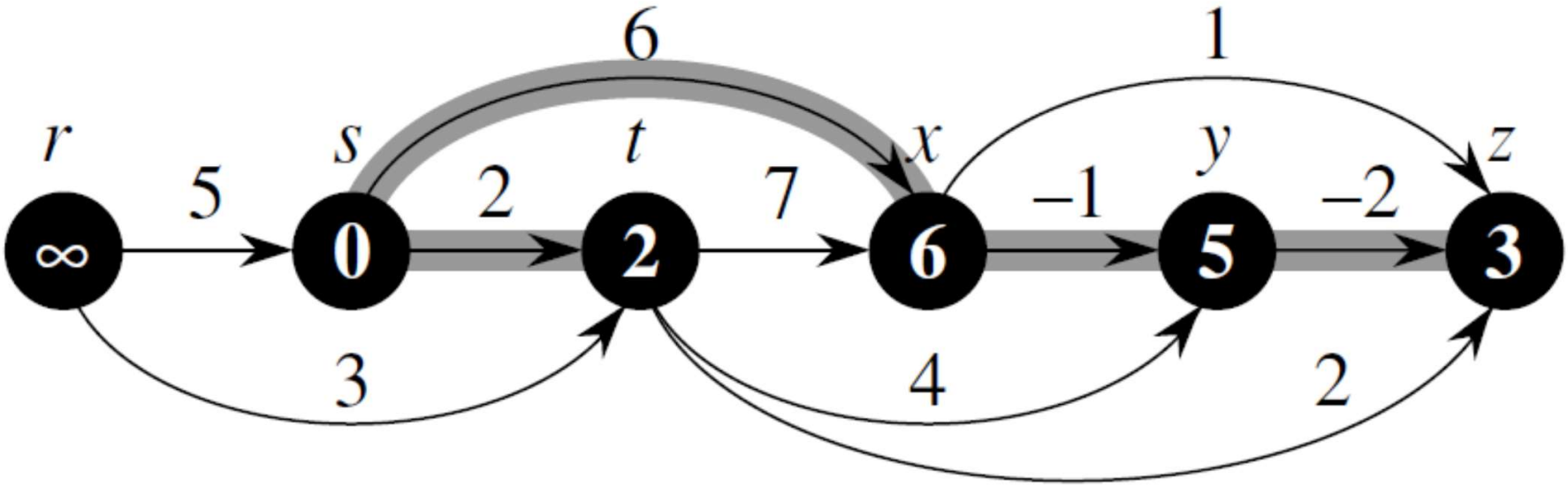
4 **do for** each vertex $v \in Adj[u]$

5 **do** RELAX(u, v, w)

Example of Single-source shortest paths in DAG



Example of Single-source shortest paths in DAG



Dijkstra’s algorithm

Aim of Dijkstra's algorithm

- Dijkstra's algorithm solves the **single-source shortest-paths** problem on a **weighted**, **directed** graph $G = (V, E)$ for the case in which all edge weights are **nonnegative**.

- The running time of Dijkstra's algorithm is **lower** than that of the Bellman-Ford algorithm

Dijkstra's algorithm - idea

- Dijkstra's algorithm maintains a **set S of vertices** whose final shortest-path weights from the source s have **already been determined**.
- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate,
  - adds u to S, and
  - relaxes all edges leaving u.

  - min-priority queue Q of vertices can be used!

DIJKSTRA - Pseudocode

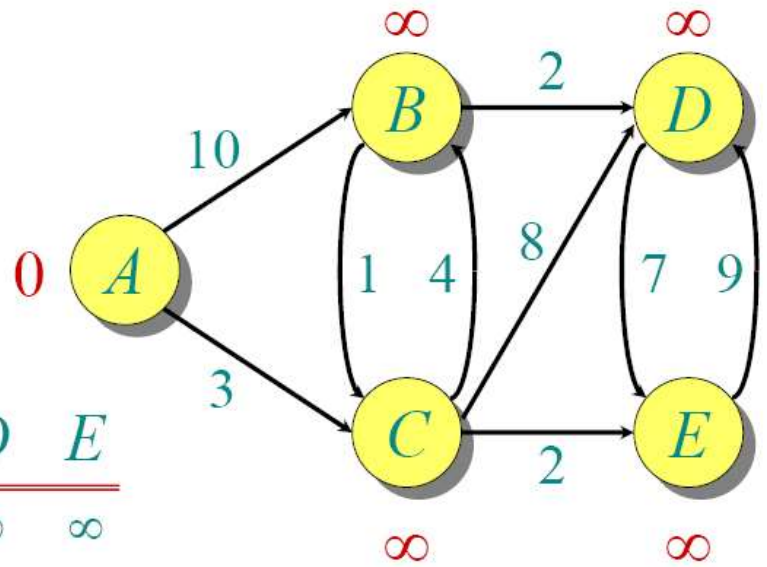
DIJKSTRA$(\boldsymbol{G}, \boldsymbol{w}, \boldsymbol{s})$

1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$ (based on d value)
4 **while** $Q = \emptyset$
5 **do** $u \leftarrow EXTRACT - MIN(Q)$
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adj[u]$
8 **do** RELAX(u, v, w)

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

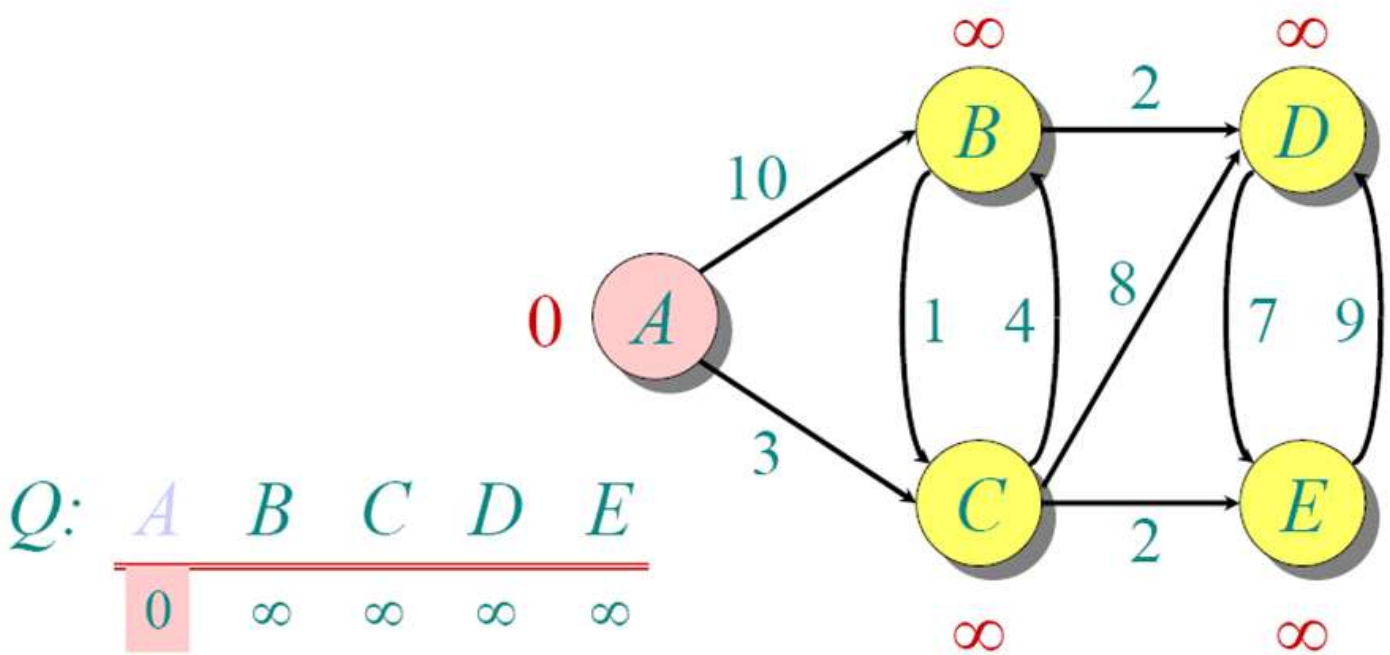
Dijkstra Example

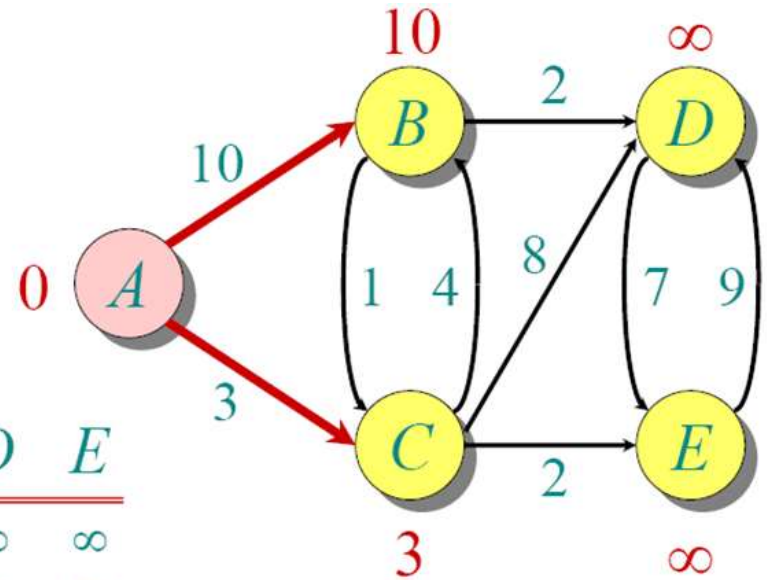
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

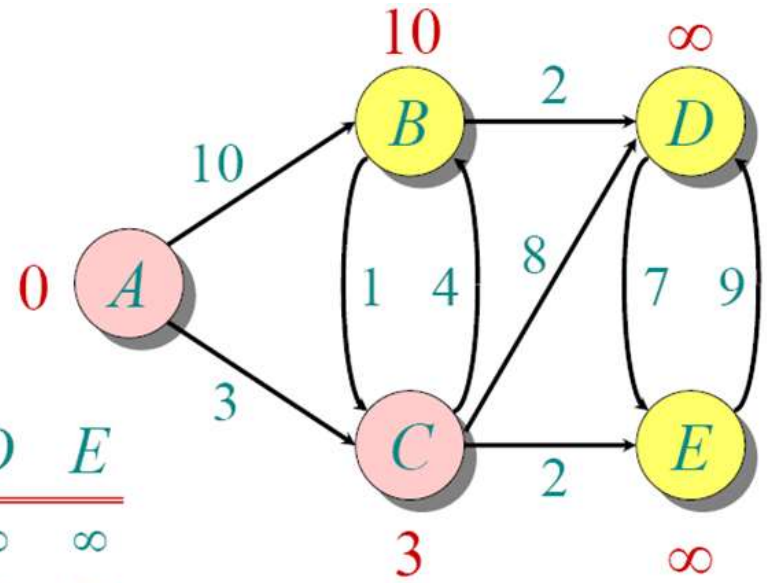
S: {}


∞
∞
B
2
D
10
0
A
1
4
8
7
9
3
C
2
E
∞
∞
Q: A B C D E
0 ∞ ∞ ∞ ∞



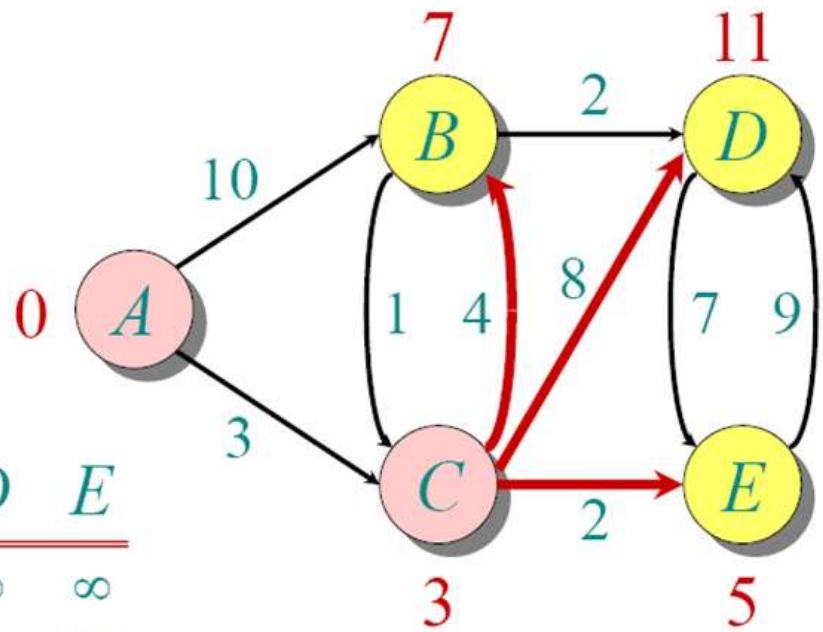
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }



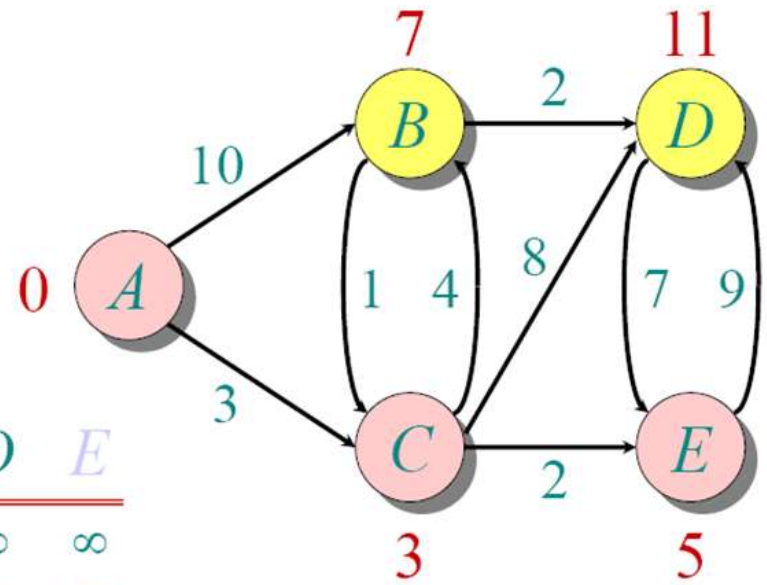
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }



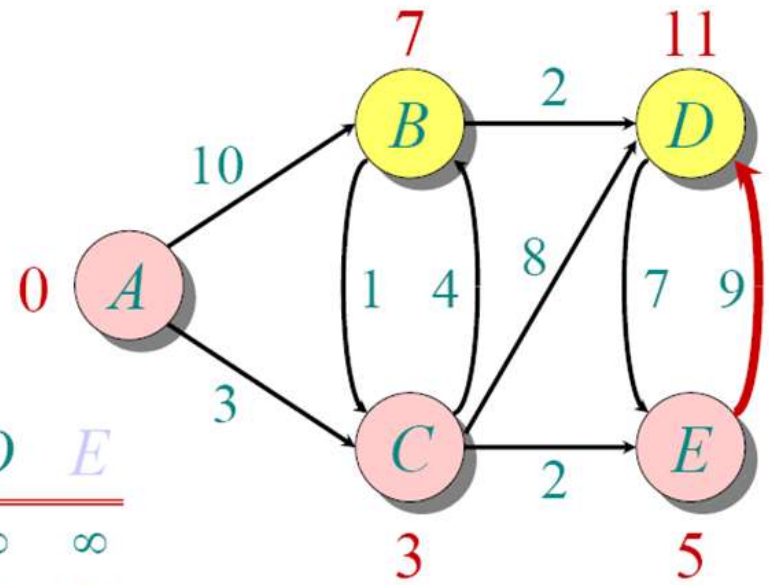
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C }



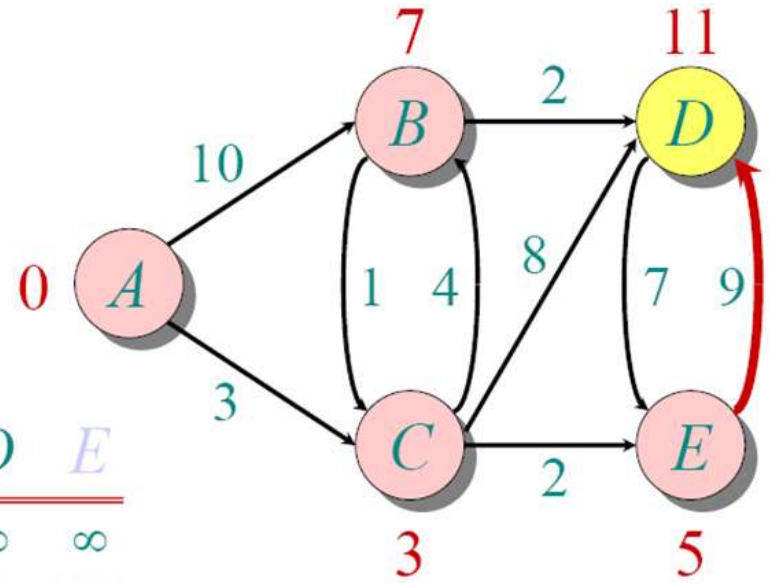
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }



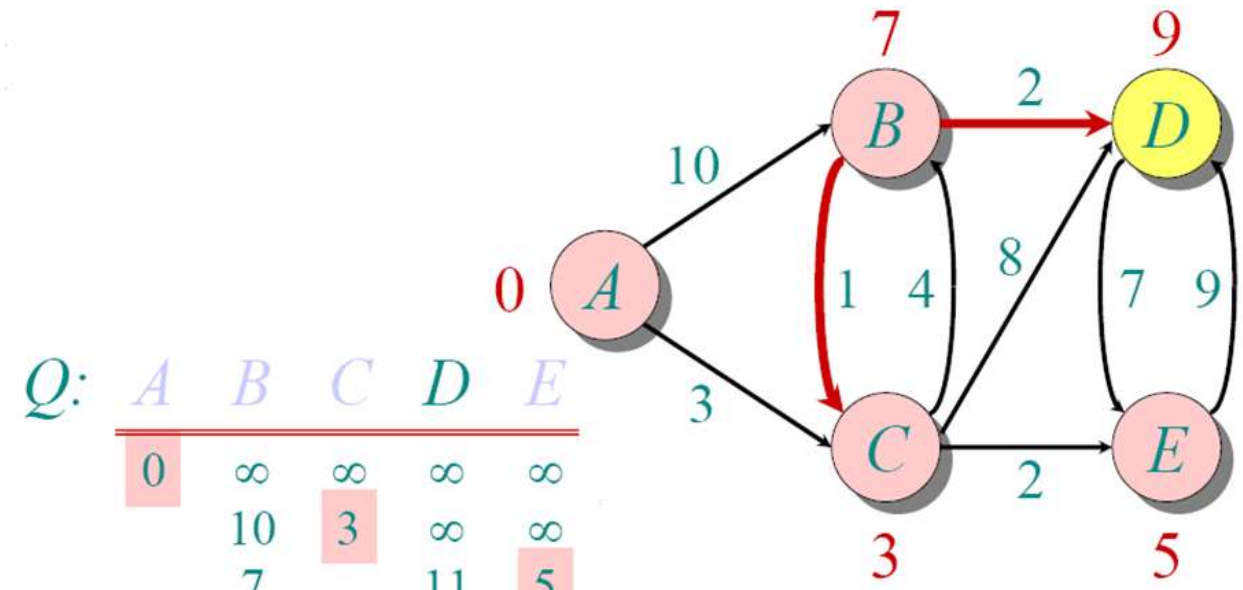
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }



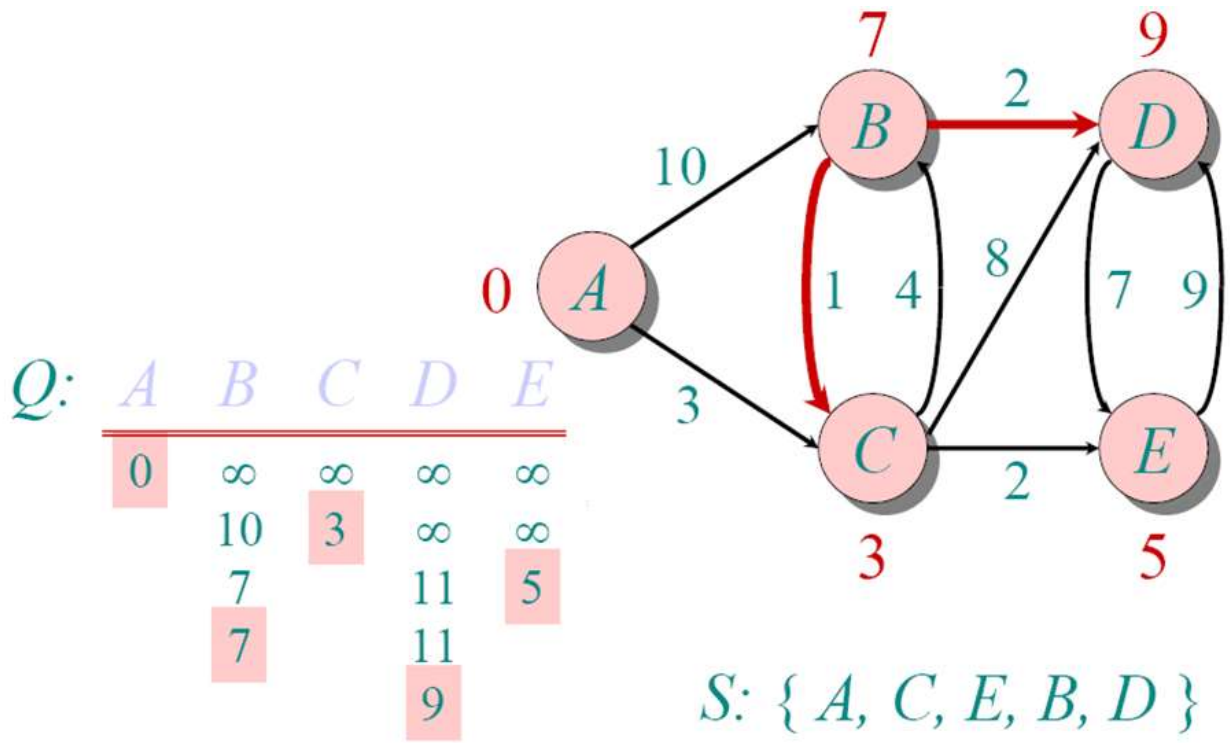
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }


7
9
2
B
D
10
0
A
1
4
8
7
9
Q:
A
B
C
D
E
3
C
E
2
3
5
0
∞
∞
∞
∞
10
3
∞
∞
7
11
5
7
11
9
S: { A, C, E, B, D }

Algorithm properties

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

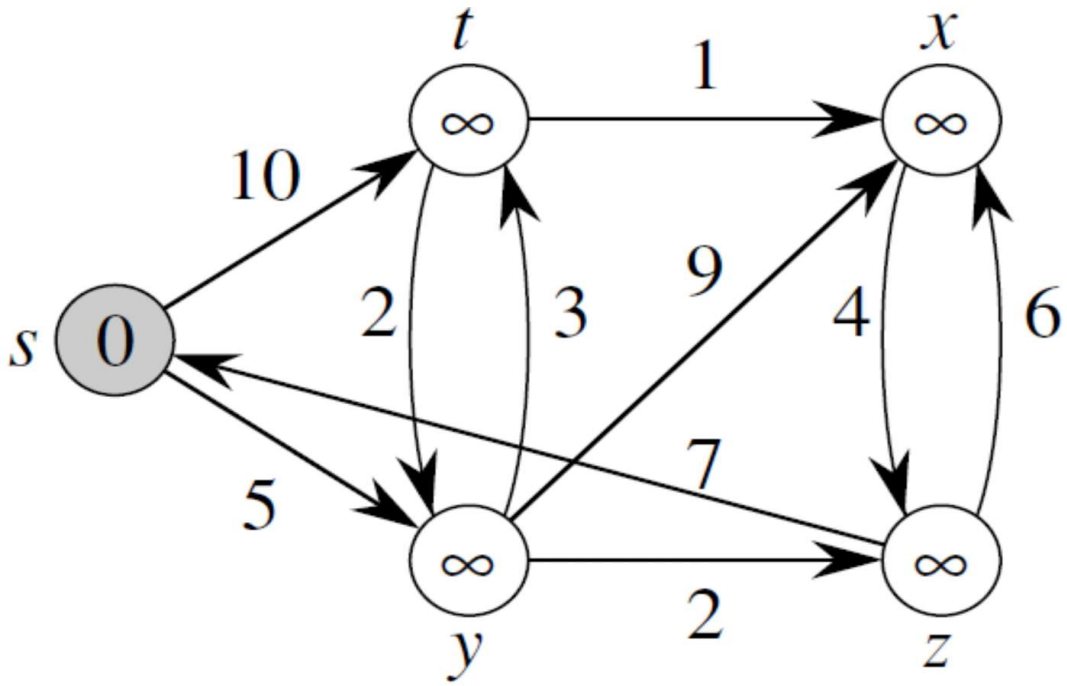
The *while* loop of lines 4–8 iterates exactly $|V|$ times.

Line 8 is executed O(|E|) times

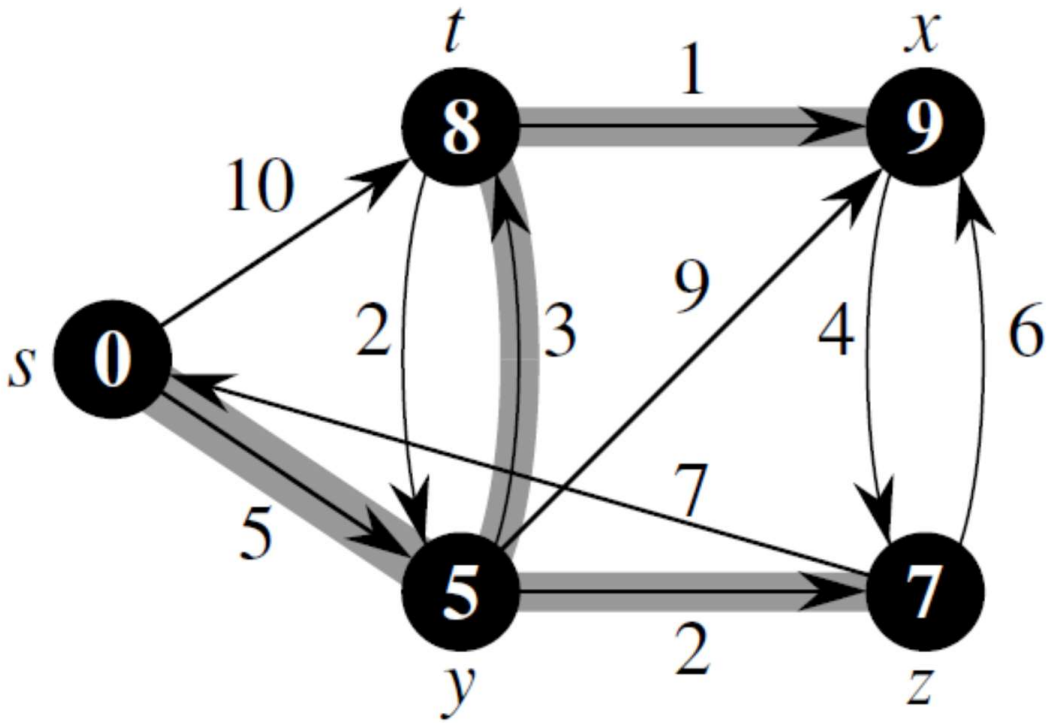
- it uses a greedy strategy.
  - always chooses the "closest" vertex in $V - S$ to add to set S

Run Dijkstra for following gragh?

write **Q** (including elements with their values) in each while iteration(there are 5 cases) and the output (**π and d**).For example, at first Q is s(0),t(∞), x(∞), y(∞), z(∞).



Expected result

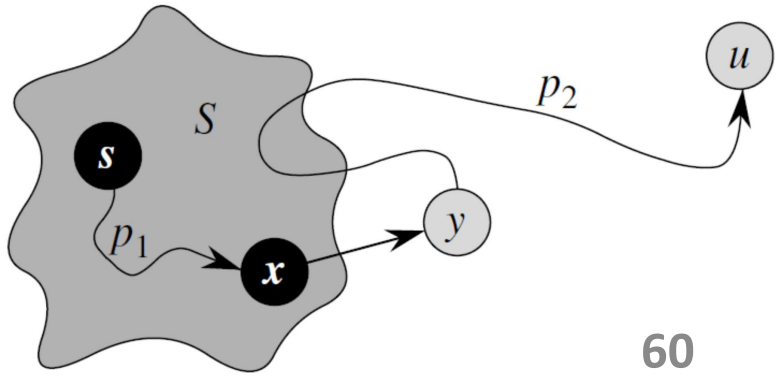


Theorem 24.6 (Correctness of Dijkstra's algorithm)

- Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source s, terminates with $d[u] = \delta(s, u)$ for all vertices $u \in V$.

- ***Proof*** the following loop invariant:
  - At the start of each iteration of the **while** loop of lines 4–8, $d[v] = \delta(s, v)$ for each vertex $v \in S$.

  -Show equality holds all the time!



Analysis

- Dijkstra maintains the min-priority queue Q by calling three priority-queue operations:
  - INSERT -> is invoked once per vertex
  - EXTRACT-MIN -> is invoked once per vertex
  - DECREASE-KEY -> at most $|E|$ by using aggregate analysis

Analysis (1) - ordinary array-$O(V^2)$

- Dijkstra maintains the min-priority queue Q by calling three priority-queue operations:
  - INSERT -> is invoked once per vertex $\boldsymbol{O(1) \rightarrow O(V \times 1)}$
  - EXTRACT-MIN -> is invoked once per vertex $\boldsymbol{O(V) \rightarrow O(V \times V)}$
  - DECREASE-KEY -> at most $|E|$ by using aggregate analysis $\boldsymbol{O(1) \rightarrow O(E \times 1)}$

- Implementing the min-priority queue: **ordinary array**
  - Number vertices from 1 to v
  - Store $d[v]$ in the vth entry of array
  - total time of $O(V^2 + E) = O(V^2)$.

Analysis (2) - min-Heap-$O(E \lg V)$.

- Dijkstra maintains the min-priority queue Q by calling three priority-queue operations:
  - INSERT -> is invoked once per vertex $\boldsymbol{O(V)(overall)}$
  - EXTRACT-MIN -> is invoked once per vertex $\boldsymbol{O(lg\ V) \rightarrow O(V\ \times lg\ V)}$
  - DECREASE-KEY -> at most $|E|$ by using aggregate analysis $\boldsymbol{O(lg\ V) \rightarrow O(E\ \times lg\ v)}$

- Implementing the min-priority queue: **min-Heap**
  - total time of $O((V + E) \lg V) = O(E \lg V)$.

- *This running time is an improvement over the straightforward $O(V^2)$ time implementation if $E = o(V^2/lg\ V)$.*

Analysis (3) - Fibonacci heap-$O(V \lg V + E)$

- Dijkstra maintains the min-priority queue Q by calling three priority-queue operations:
  - INSERT -> is invoked once per vertex $\boldsymbol{O(V)(overall)}$
  - EXTRACT-MIN -> is invoked once per vertex amortized $\boldsymbol{O(lg\ V) \rightarrow O(V\ \times lg\ v)}$
  - DECREASE-KEY -> at most $|E|$ by using aggregate analysis amortized $\boldsymbol{O(1) \rightarrow O(E\ \times 1)}$

- Implementing the min-priority queue: **Fibonacci heap**
  - total time of $O(V \lg V + E)$