Greedy Algorithm

- A technique to solve problems:
  - always makes the locally best choice at the moment (local optimal).
  - Hopefully, a series of locally best choices will lead to a globally best solution.
- Greedy algorithms yield optimal solutions for many (but not all) problems.

Ideas for Interval Scheduling and Interval Partitioning

sort all the jobs in a list using a 'greedy' principle, and then choose it one by one.

Kruskal’s Algorithm

- In the beginning, we have a forest where each node is a tree.
- merge these trees step by step.

  We connect two trees with an edge. This edge is chosen to be a shortest one connecting two different trees.
- Eventually, the forest becomes a tree.

Prim’s algorithm

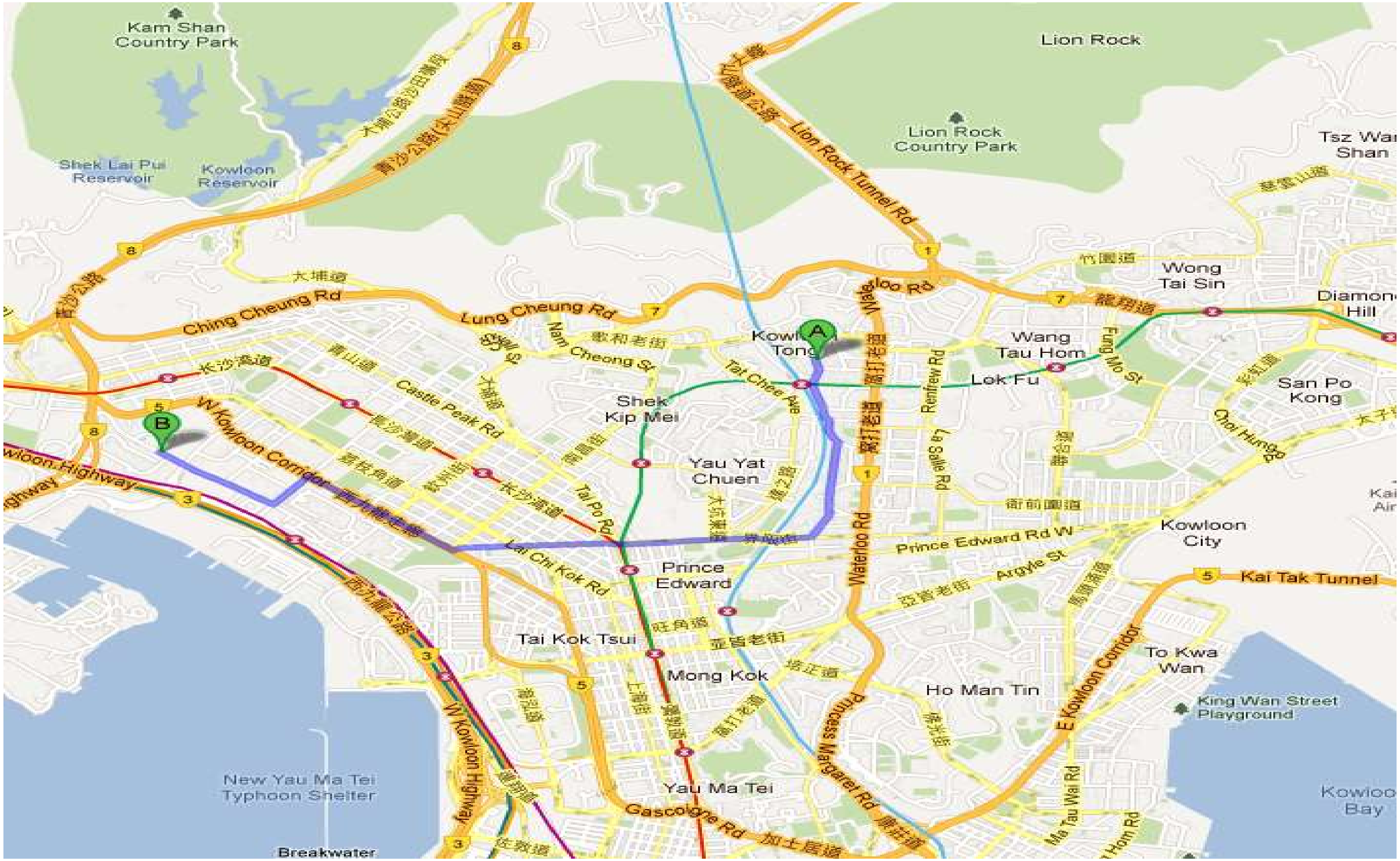
In the beginning, Set T as a single node.

At each step, add to T the shortest edge between a node in T and a node not in T.

Finally, T becomes a spanning tree.

Single-Source Shortest Paths

- Problem Definition
- Shortest paths and Relaxation
- Dijkstra's algorithm (a greedy algorithm)



Find a shortest path from A to B.

-need serious thinking to get a correct algorithm.

Problem Definition:

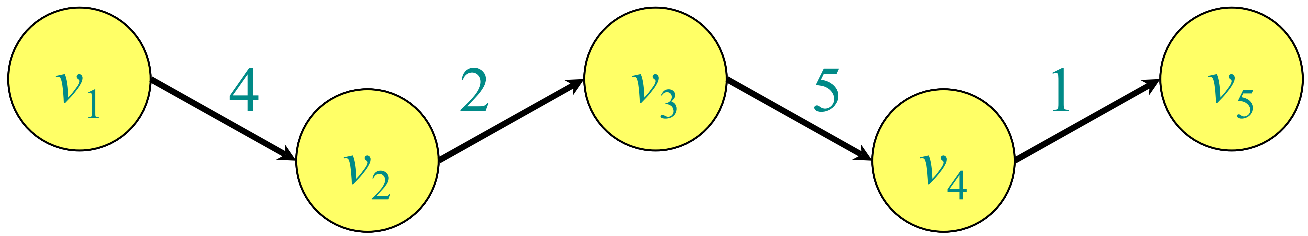
- Given a directed graph $G=(V, E, W)$, where each edge has a weight (length, cost),
- Find a shortest path from s to v.
  - s—source
  - v—destination.

Paths in graphs

Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to R$. The total weight (length) of path $p = v_1 \to v_2 \to \ldots. \to v_k$ is defined to be

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

Example:



$w(p) = 4+2+5+1=12$

Shortest paths

A ***shortest path*** from u to v is a path of minimum total weight from u to v. The ***shortest-path weight*** from u to v is defined as

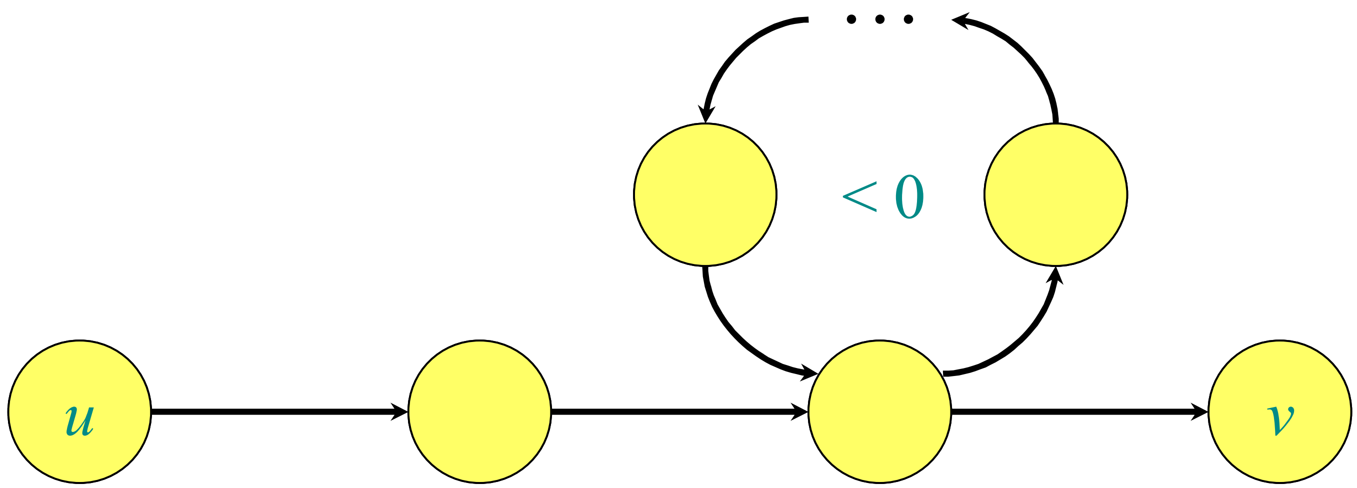
$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.$$

Note: $\delta(u, v) = +\infty$ if no path from u to v exists.

Well-definedness of shortest paths

If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Example:



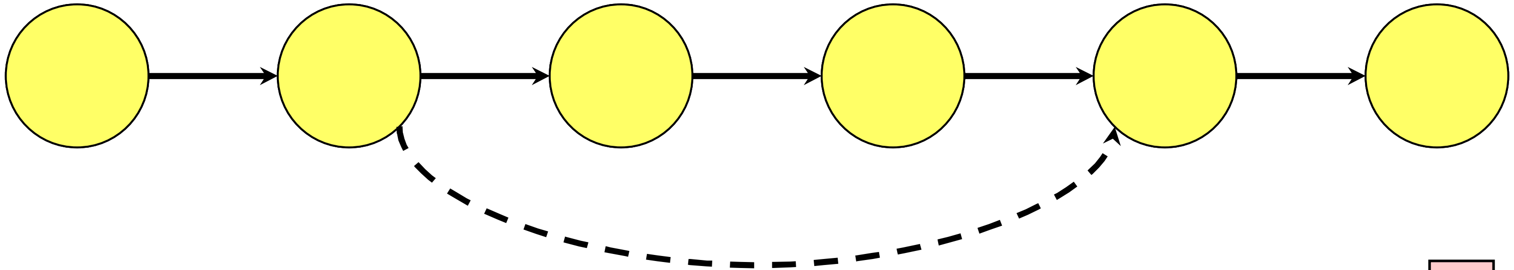
Negative-Weight edges:

- negative-weight cycles
  - the total weight in the cycle (circuit) is negative.
- If no negative-weight cycles reachable from the source s, then for all v $\in$V, the shortest-path weight remains well defined, even if it has a negative value.
- If there is a negative-weight cycle on some path from s to v, we define $\delta(s,v) = -\infty$.
- We assume that all the edges have weights >=0.

Optimal substructure

Theorem. A subpath of a shortest path is a shortest path.

Outline of Proof.



$\square$

Representing shortest paths:

- we maintain for each node $v \in V$, a **predecessor** π *[v]* that is the vertex in the shortest path right before v.
- With the values of π, a backtracking process can give the shortest path. *(example)*

- Observation: (basic)
- Suppose that a shortest path p from a source s to a vertex v can be decomposed into

$$s \xrightarrow{p'} u \rightarrow v$$

for some vertex u and path p'. Then, the weight of a shortest path from s to v is

$$\delta(s,v) = \delta(s,u) + w(u,v)$$

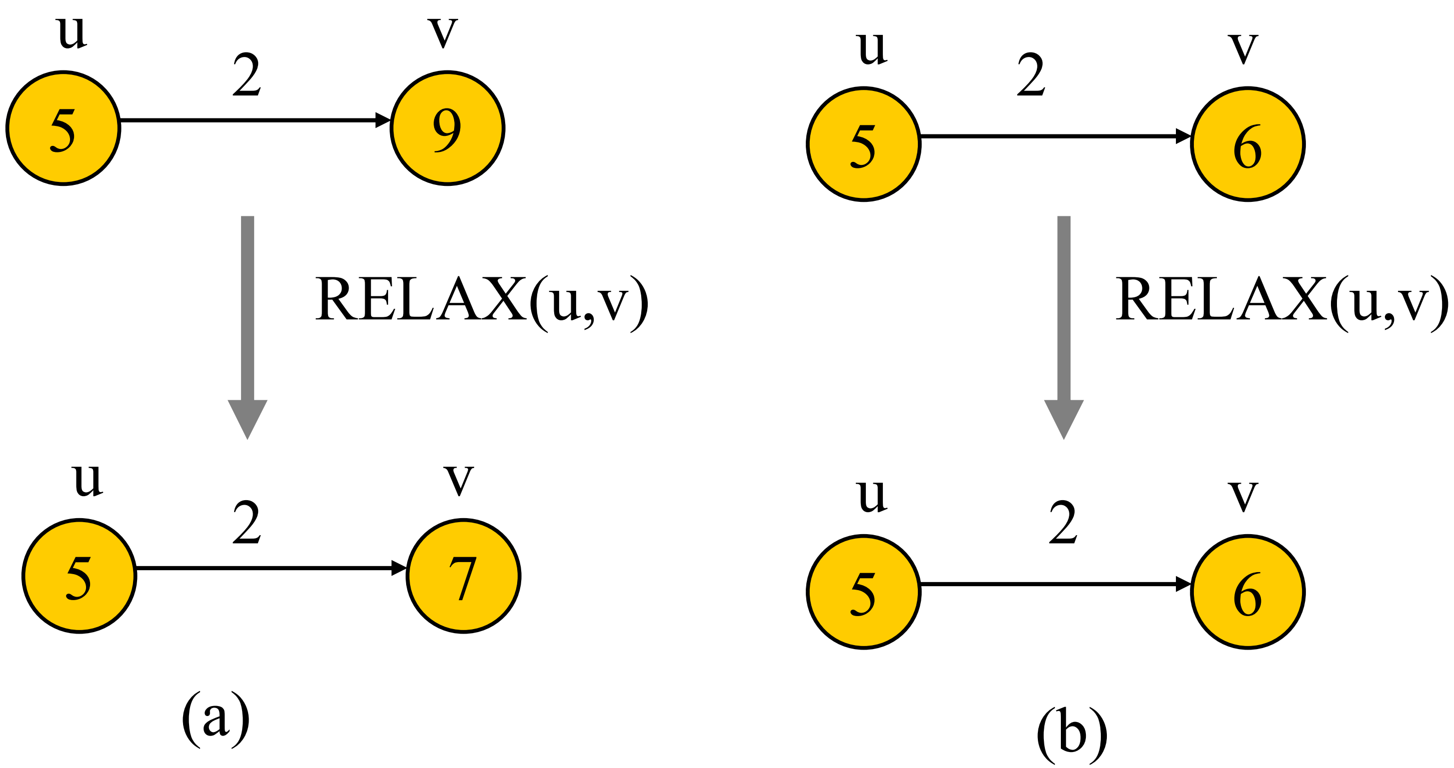
We do not know what is u for v, but we know u is in V and we can try all nodes in V in O(n) time.

Also, if u does not exist, the edge (s, v) is the shortest.

Question: how to find (s, u), the first shortest from s to some node?

Relaxation

- $\pi[v]$: predecessor of v, d[v]: the total weight of path from 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].
- RELAX(u, v, w)
- if d[v]>d[u]+w(u, v)
- then d[v] $\longleftarrow$ d[u]+w(u,v) *(based on observation)*
- π[v] $\longleftarrow$ u



relaxation of an edge (u,v).The shortest-path estimate of each vertex is shown within the vertex. (a)Because d[v]>d[u]+w(u,v) prior to relaxation, the value of d[v] decreases. (b)Here, d[v] $\leq$ d[u]+w(u,v) before the relaxation step,so d[v] is unchanged by relaxation.

Single-source shortest paths

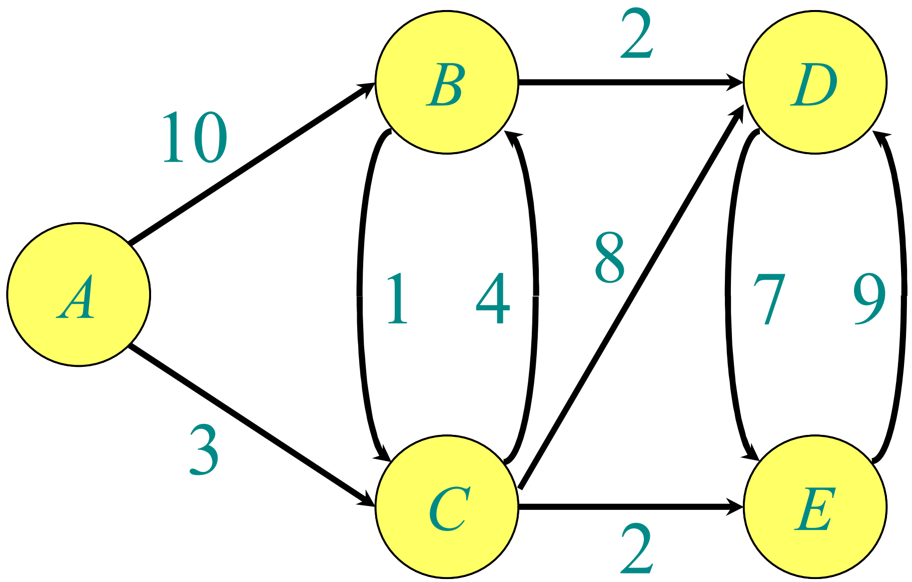
Problem. From a given source vertex $s \in V$, find a shortest-path from s to every $v \in V$.

IDEA: Greedy

Maintain S, π and d.

- ***S***: ***the set* of nodes whose shortest-paths from *s* have been found**.
- For every vertex $v \in V$, $\pi(v) \in S, or\ \ \pi(v) = NIL$.
  π defines the shortest path from s to every node only using nodes in S as intermediate nodes. d: the weight (length) of this shortest path.
- At each step add to S the vertex v in $V - S$ whose d value is minimal.
- Update the d and π values of all the nodes (in V-S) adjacent to v.

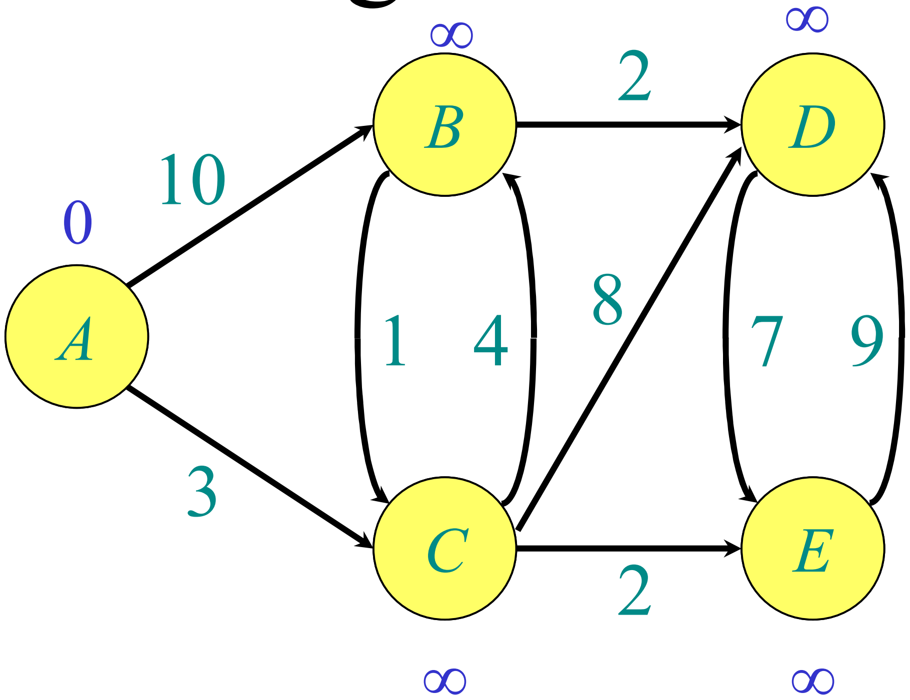
Example of Dijkstra's algorithm



Example of Dijkstra's algorithm

Initialize:

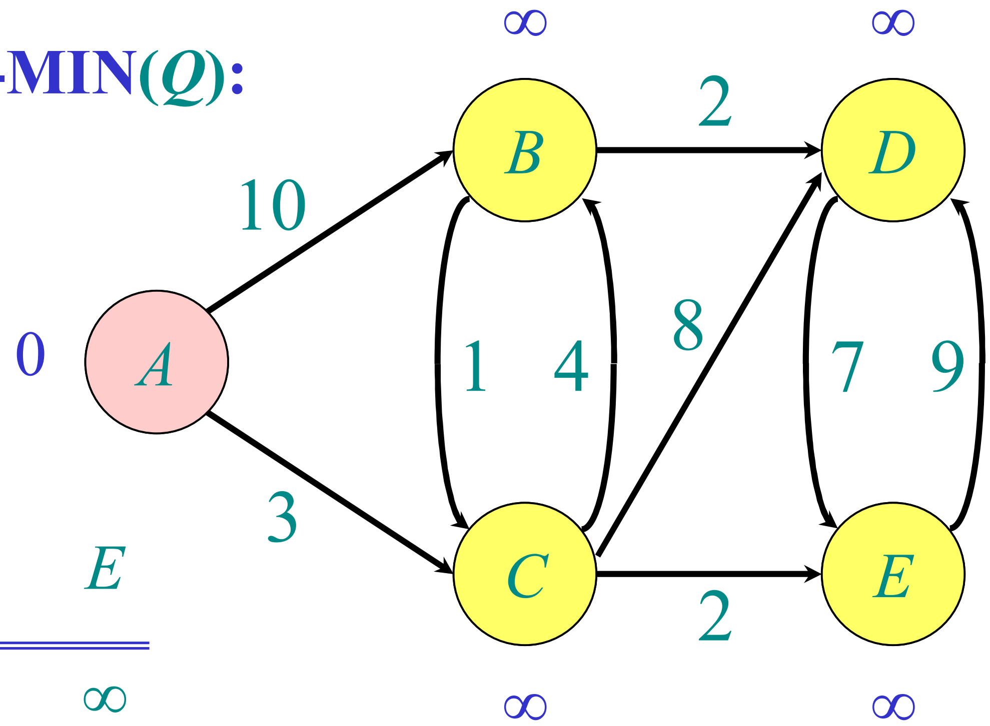
Q:	A	B	C	D	E
d	0	∞	∞	∞	∞
π	Nil	Nil	Nil	Nil	Nil



S: {}

Example of Dijkstra’s algorithm

“*A*” ← EXTRACT-MIN(*Q*):



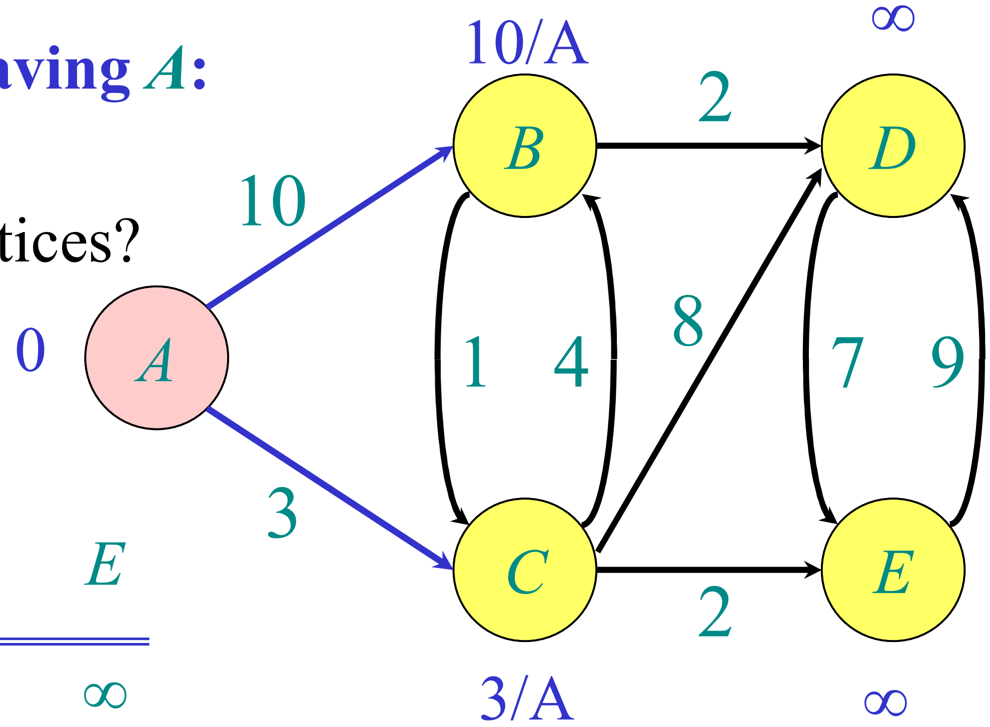
Q:	A	B	C	D	E
d	0	∞	∞	∞	∞
π	*NIL*	*NIL*	*NIL*	*NIL*	*NIL*

S: { *A* }

Example of Dijkstra's algorithm

Relax all edges leaving *A*:

Q: can A be a better predecessor for other vertices?

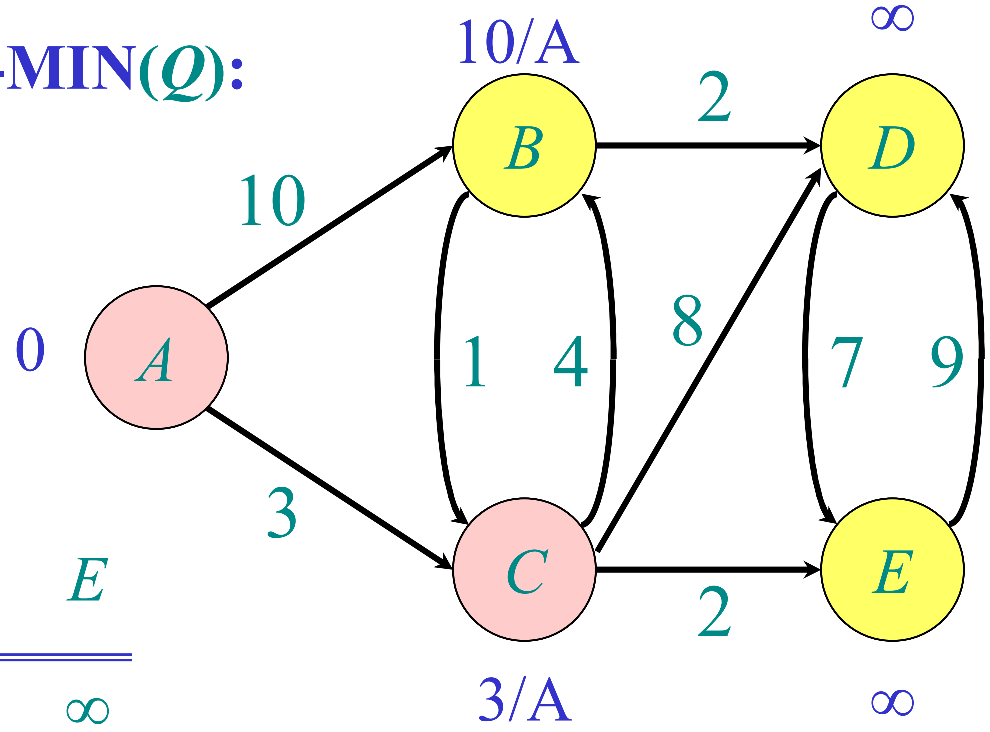


Q:	*A*	*B*	*C*	*D*	*E*
d	0	10	3	∞	∞
π	*NIL*	*A*	*A*	*NIL*	*NIL*

S: { *A* }

Example of Dijkstra's algorithm

"C" ← EXTRACT-MIN(Q):

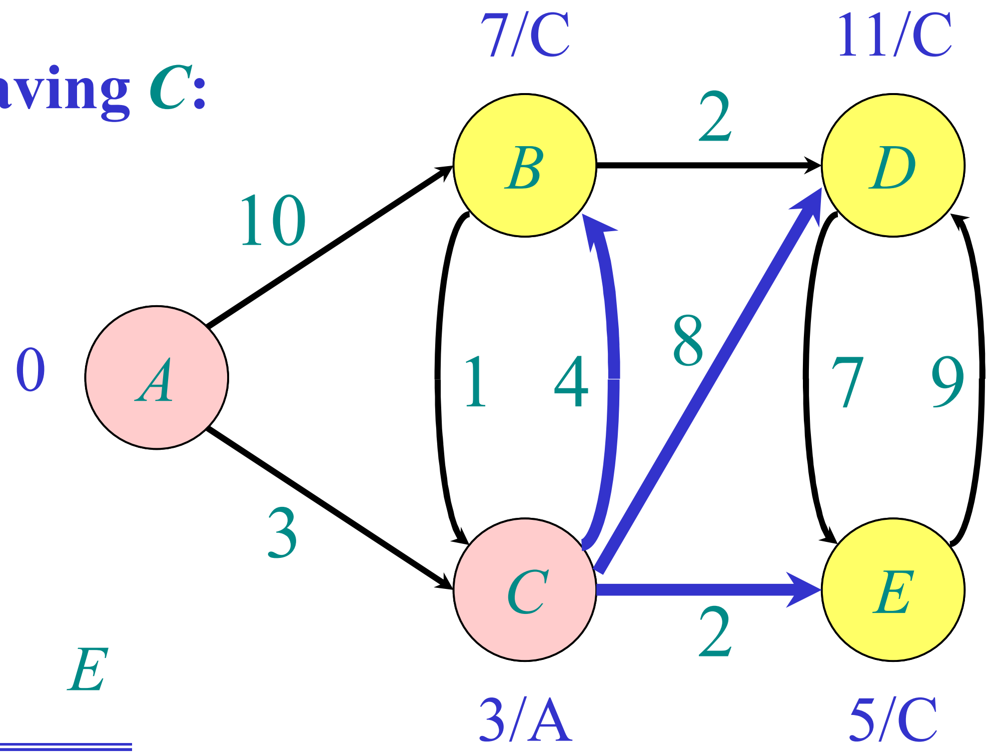


Q:	A	B	C	D	E
d	0	10	3	∞	∞
π	*NIL*	*A*	*A*	*NIL*	*NIL*

S: { A, C }

Example of Dijkstra's algorithm

Relax all edges leaving *C*:

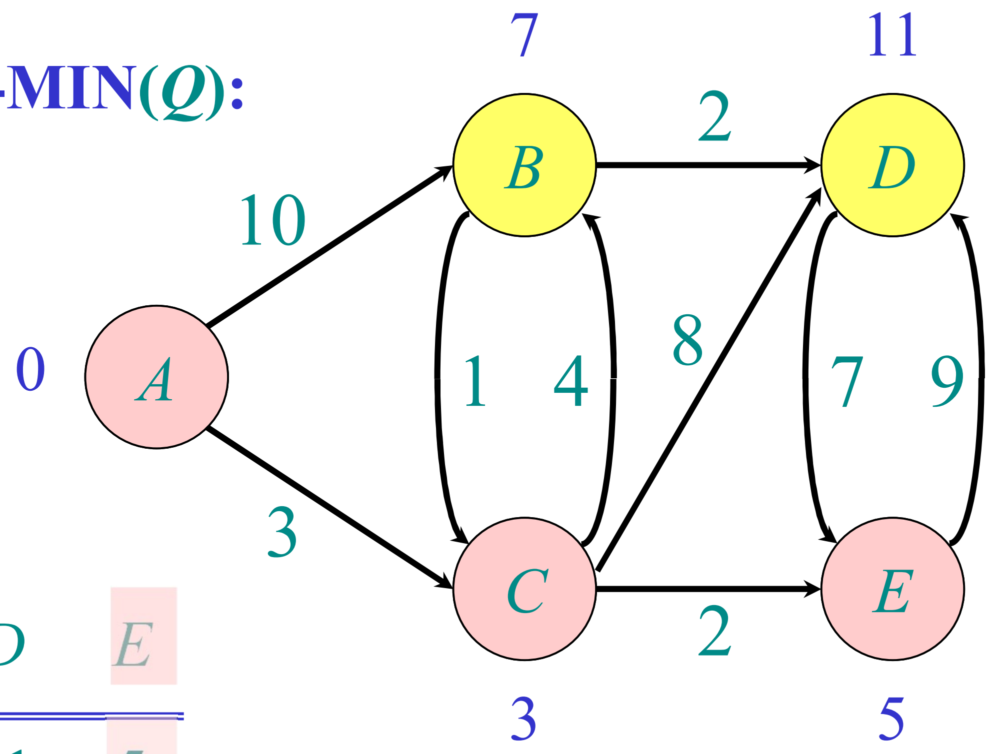


Q:	*A*	*B*	*C*	*D*	*E*
d	0	7	3	11	5
π	*NIL*	*C*	*A*	*C*	*C*

S: { *A*, *C* }

Example of Dijkstra's algorithm

"E" ← EXTRACT-MIN(Q):

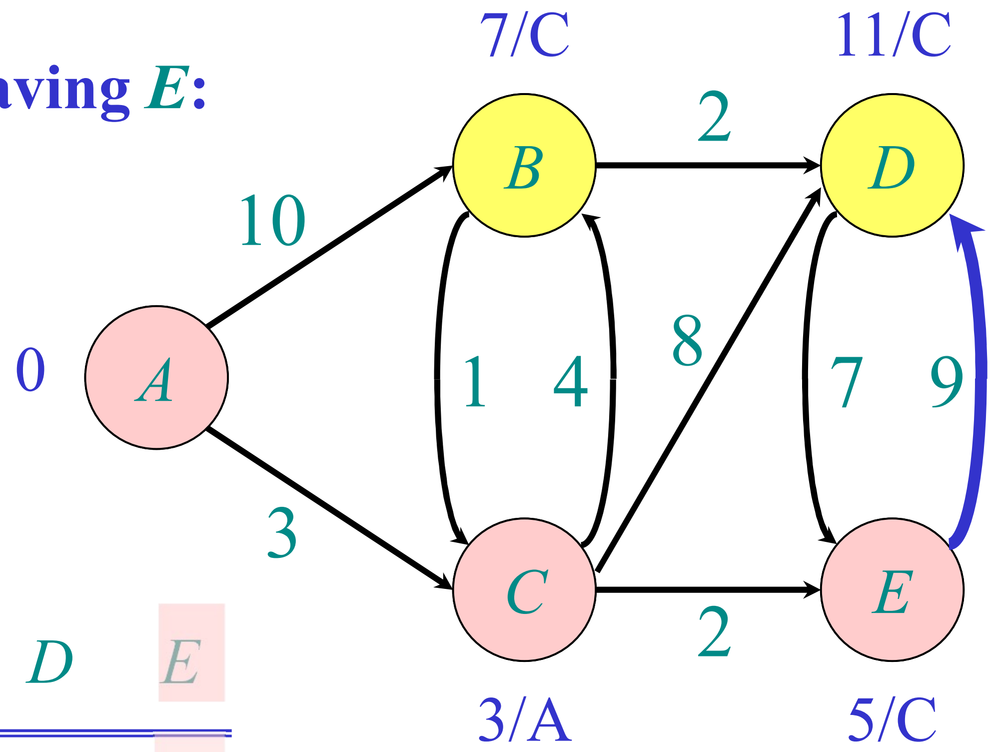


Q:	A	B	C	D	E
d	0	7	3	11	5
π	NIL	C	A	C	C

S: { A, C, E }

Example of Dijkstra's algorithm

Relax all edges leaving *E*:

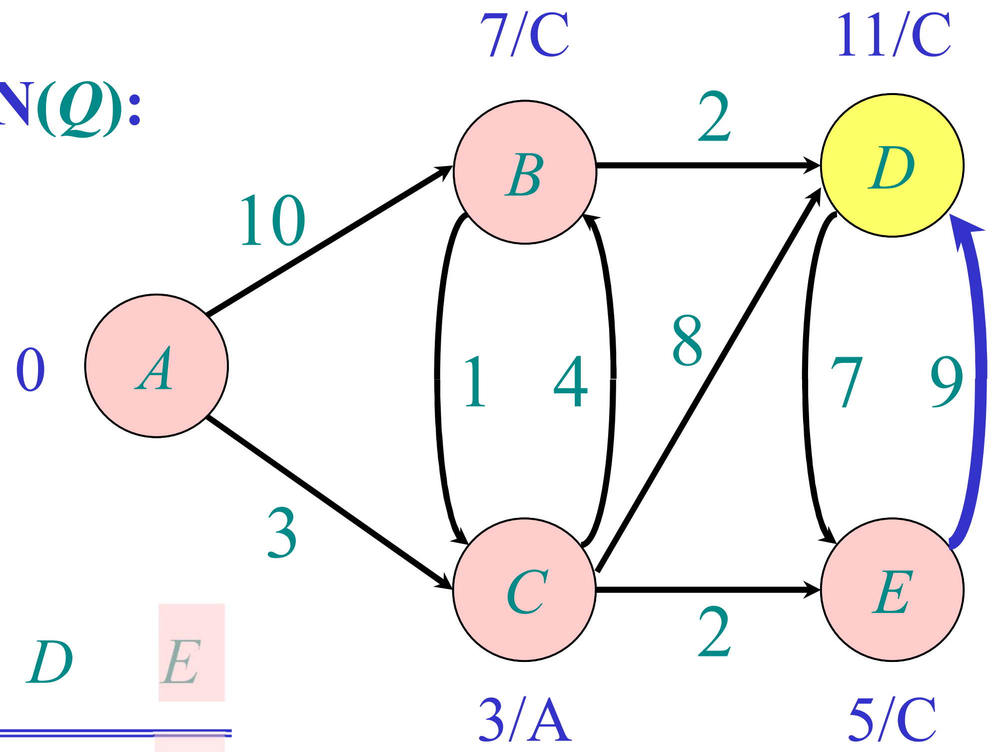


Q:	*A*	*B*	*C*	*D*	*E*
d	0	7	3	11	5
π	*NIL*	*C*	*A*	*C*	*C*

S: { *A*, *C*, *E* }

Example of Dijkstra's algorithm

"*B*" ← EXTRACT-MIN(*Q*):

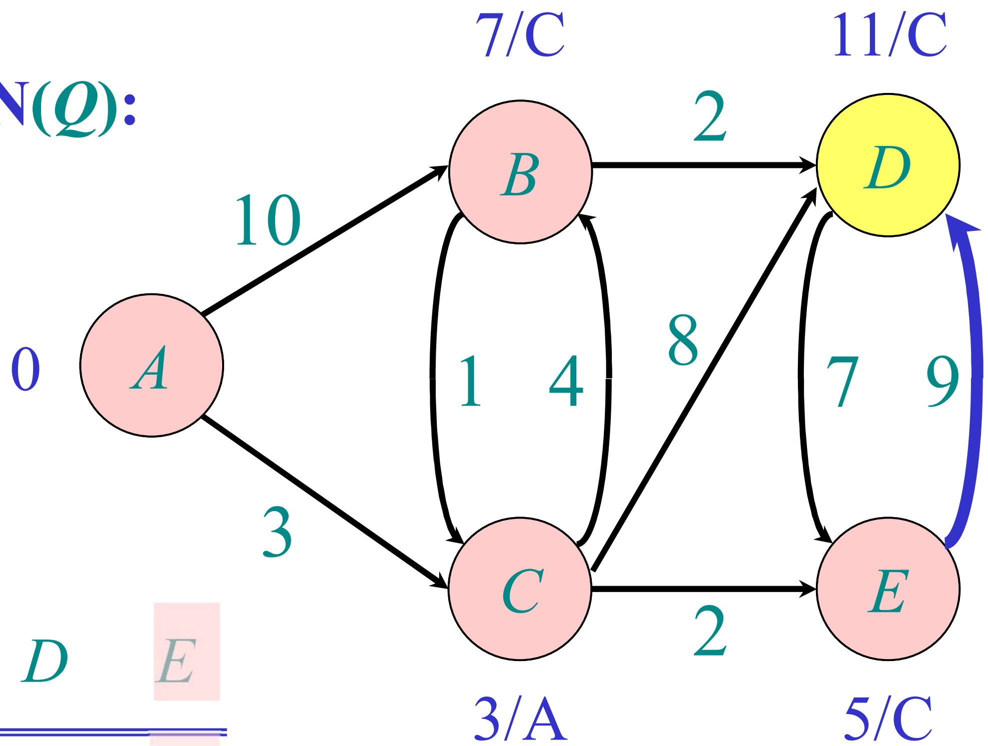


Q:	*A*	*B*	*C*	*D*	*E*
d	0	7	3	11	5
π	*NIL*	*C*	*A*	*C*	*C*

S: { *A*, *C*, *E*, *B* }

Example of Dijkstra's algorithm

"*B*" ← EXTRACT-MIN(*Q*):

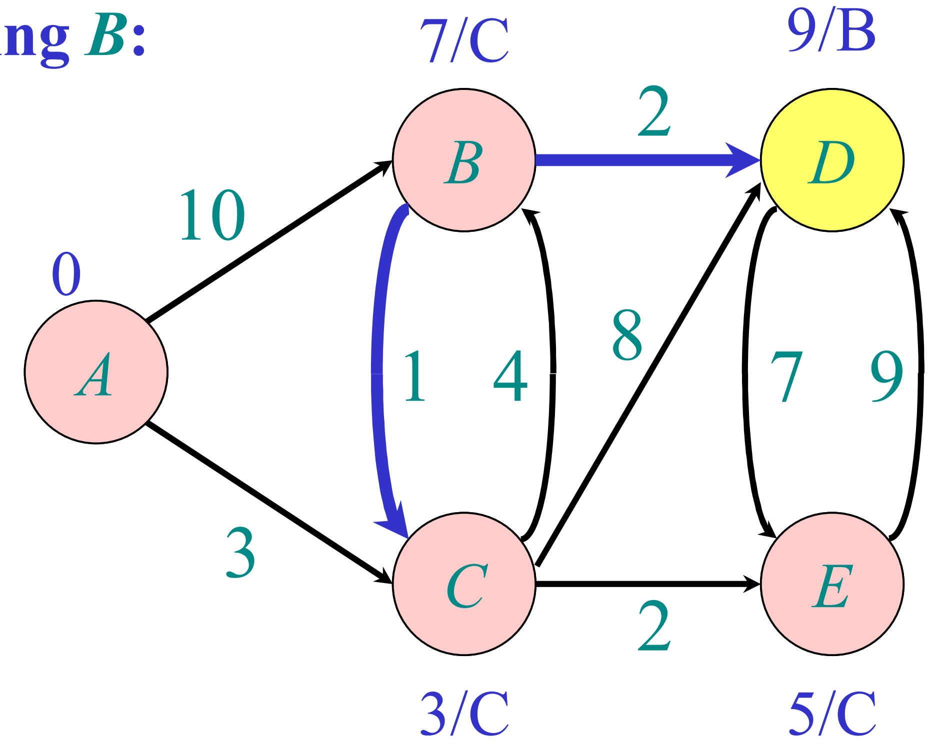


Q:	*A*	*B*	*C*	*D*	*E*
d	0	7	3	11	5
π	*NIL*	*C*	*A*	*C*	*C*

S: { *A*, *C*, *E*, *B* }

Example of Dijkstra's algorithm

Relax all edges leaving *B*:



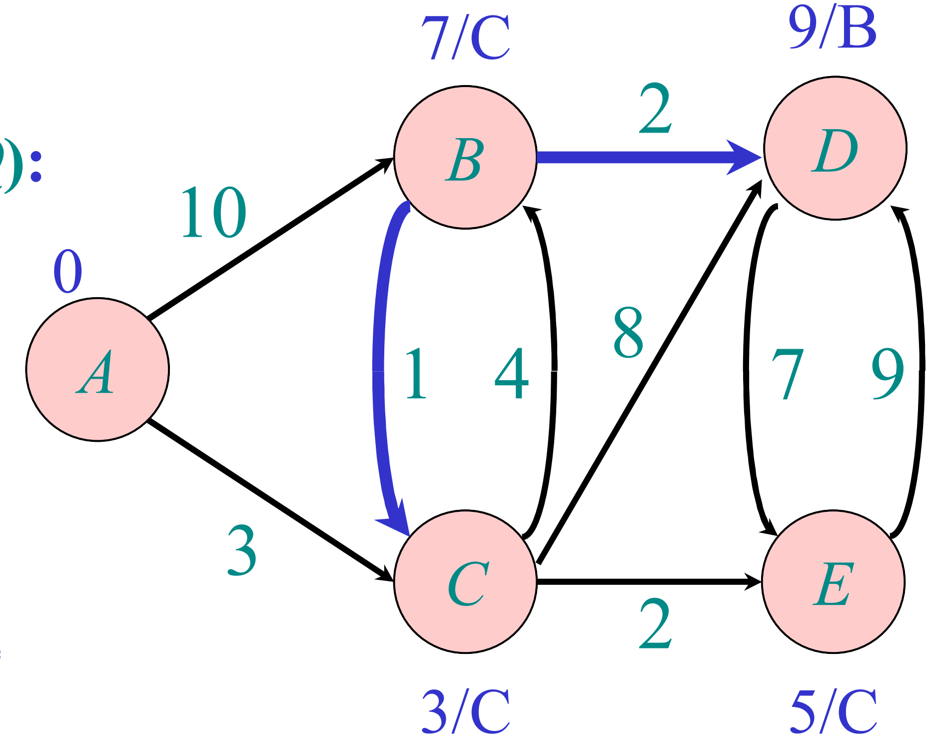
Q:	*A*	*B*	*C*	*D*	*E*
d	0	7	3	9	5
π	*NIL*	*C*	*A*	*B*	*C*

S: { *A*, *C*, *E*, *B* }

Example of Dijkstra's algorithm

***"D"* ← EXTRACT-MIN(*Q*):**

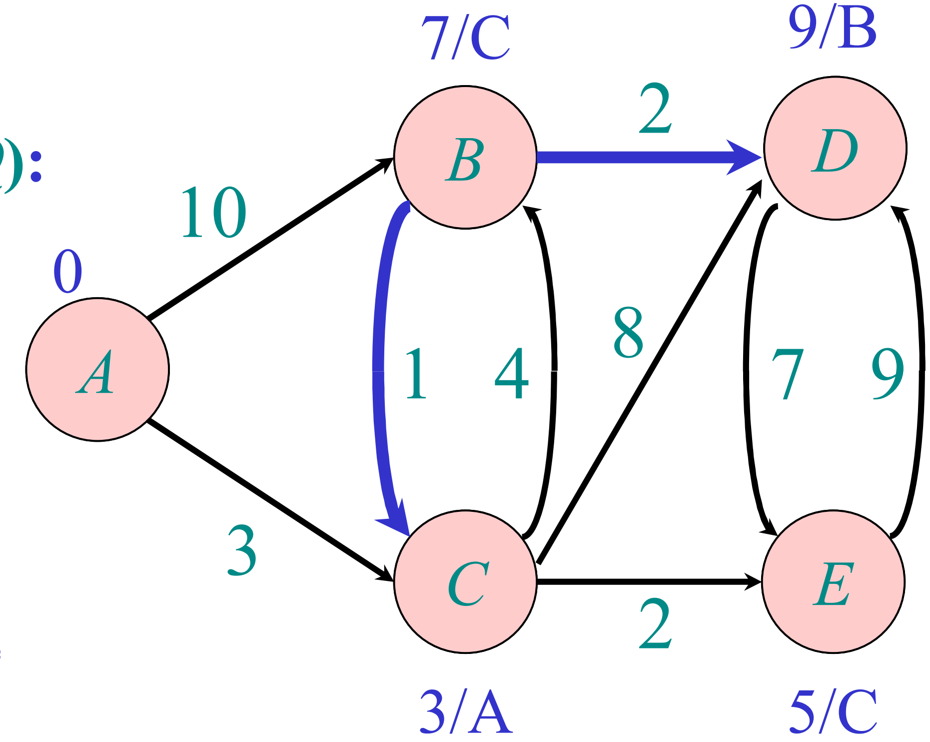
Q:	*A*	*B*	*C*	*D*	*E*
d	0	7	3	9	5
π	*NIL*	*C*	*A*	*B*	*C*



S: { *A*, *C*, *E*, *B*,*D* }

Example of Dijkstra’s algorithm

“*D*” ← EXTRACT-MIN(*Q*):



Q:	*A*	*B*	*C*	*D*	*E*
d	0	7	3	9	5
π	*NIL*	*C*	*A*	*B*	*C*

S: { *A*, *C*, *E*, *B*,*D* }

Backtracking to find the shortest path from A to D:
D←B ← C ← A

Backtracting code: print out the path from s to u.

```
print (u)
x=π(u)
while (x ≠ s)
    print (x)
    x=π(x)
print x
```

Initialization:

- For each vertex v $\in$ V, d[v] denotes the length of the shortest path found so far from source s to v.
- d[v]– will be δ(s, v) after the execution of the algorithm.
- initialize d[v] and π[v] as follows: .
- INITIALIZE-SINGLE-SOURCE(G,s)
- for each vertex v $\in$ V\{s}
- do d[v] $\longleftarrow \infty$
- π[v] $\longleftarrow$NIL
- d[s] $\longleftarrow$0

Dijkstra's Algorithm:

- Dijkstra's algorithm assumes that $w(e) \geq 0$ for each e in the graph.
- maintain a set S of vertices such that
  - Every vertex $v \in S$, $d[v]=\delta(s, v)$, i.e., the shortest-path from s to v has been found. (Initial values: S=empty, d[s]=0 and d[v]=∞)
- (a) select the vertex $u \in V\text{-}S$ such that

  $$d[u]=min\ \{d[x] | x \in V\text{-}S\}. \text{ Set } S=S \cup \{u\}$$

  (b) ***for*** each node v adjacent to u ***do*** RELAX(u, v, w).
- Repeat step (a) and (b) until S=V.

Dijkstra's algorithm

$d[s] \leftarrow 0$

for each $v \in V - \{s\}$

  do $d[v] \leftarrow \infty$, $\pi[v] \leftarrow NIL$.

$S \leftarrow \varnothing$

$Q \leftarrow V$ % Q is a priority queue maintaining $V - S$

while $Q \neq \varnothing$

  do $u \leftarrow$ EXTRACT-MIN(Q)

    $S \leftarrow S \cup \{u\}$

relaxation step

    for each $v \in Adj[u]$

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

        then $d[v] \leftarrow d[u] + w(u, v)$, $\pi[v] \leftarrow u$,

Implicit DECREASE-KEY using update(v,k)

Implementation:

- a priority queue Q stores vertices in V-S, keyed by their d[] values.
- the graph *G* is represented by adjacency lists.

Theorem: Consider the set S at any time in the algorithm's execution. For each $v \in S$, the current d[v] is the length of the shortest s-v path.

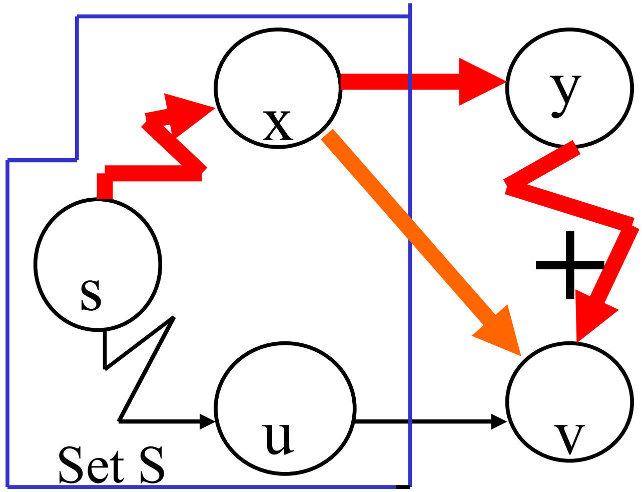
***Proof*:** We prove it by induction on |S|.

1. If |S|=1, then the theorem holds. (Because d[s]=0 and S={s}.)
2. Suppose that the theorem is true for |S|=k for some k>0.
3. Now, we grow S to size k+1 by adding the node v.

Proof: (continue)

Now, we grow S to size k+1 by adding the node v with the smallest d[v].

Let P be a path from s to v.



Case 1: All the nodes before v in P are in S. Noting that

$$d[v] = \min_{\{u \in S\}} \{d[u] + w(u, v)\}$$

Thus the length of P=d[x]+w(x,v) ≥d[v].

Case 2: P=s,…,x, y, …, v, where x is the last node in S and y is the first node not in as shown in fig.

Noting that

(i) algorithm always selects the node with the smallest value d → d[y] ≥d[v], and

(ii) any edge ≥0

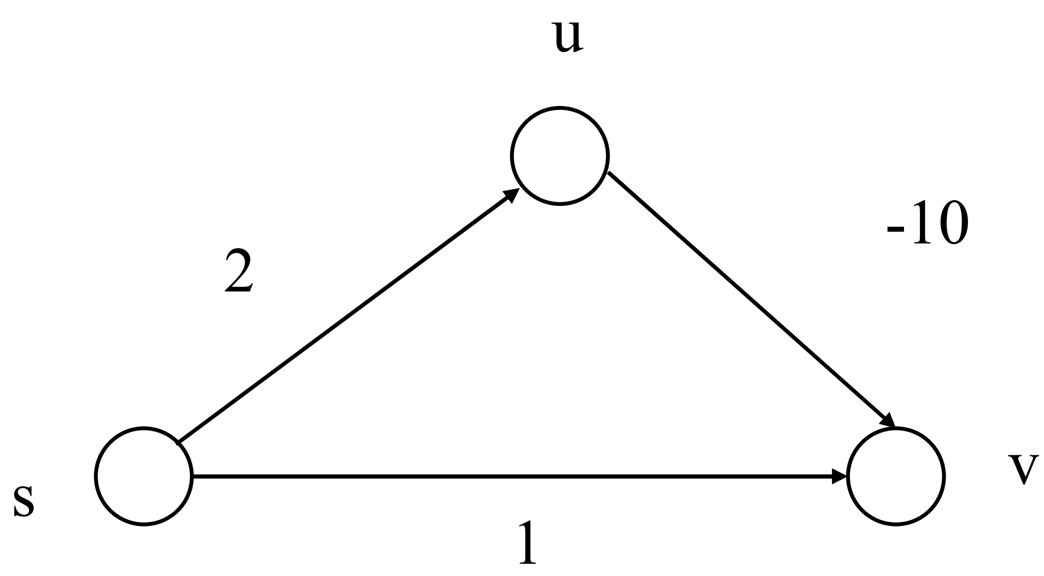
Thus, *the length of P* ≥d[y] ≥d[v].

Therefore, the current d[v] is the length of the shortest s-v path.

Time complexity of Dijkstra's Algorithm:

- Time complexity depends on implementation of the Queue.
- **Method 1:** Use an array to story the Queue
- EXTRACT -MIN(Q) --takes O(|V|) time.
  - Totally, there are |V| EXTRACT -MIN(Q)'s.
  - time for |V| EXTRACT -MIN(Q)'s is $O(|V|^2)$.
- RELAX(u,v,w) --takes O(1) time.
  - Totally |E| RELAX(u, v, w)'s are required.
  - time for |E| RELAX(u,v,w)'s is O(|E|).
- Total time required is $O(|V|^2+|E|)=O(|V|^2)$
- Backtracking with π[] gives the shortest path in inverse order.
- **Method 2:** The priority queue is implemented as a adaptable heap. It takes O(log n) time to do EXTRACT-MIN(Q) as well as | RELAX(u,v,w). The total running time is O(|E|log |V|).

The algorithm does not work if there are negative weight edges in the graph



s$\rightarrow$v is shorter than s $\rightarrow$ u, but it is longer than

s $\rightarrow$ u $\rightarrow$ v.