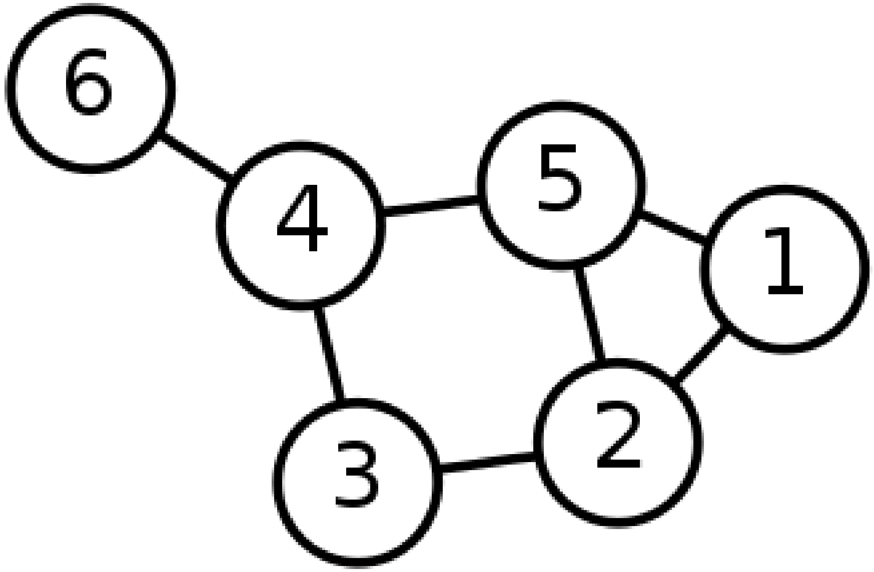
Single source shortest path algorithm

Single-Source Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Dijkstra's Algorithm
Bellmann Ford Algorithm

Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs.
Requirement: All edges must have non-negative weights.

Approach: ????

Input: Weighted graph **G={E,V}** and **source vertex s∈V**, such that all edge weights are nonnegative

Output: Lengths of shortest paths from a given source vertex *to all other vertices*

DIJKSTRA'S ALGORITHM: RELAXATION

Algorithms keep track of dist[v], P[v]. **Initialized** as follows:

```
Initialize(G, s)
    for each v ∈ V[G] do
        dist[v] := ∞;
        P[v] := NIL
    end for
    dist[s] := 0
    S := φ
```

These values are changed when an edge (u, v) is **relaxed**:

```
Relax(u, v, w)
    if dist[v] > dist[u] + w(u, v) then
        dist[v] := dist[u] + w(u, v);
        P[v] := u
    end if
```

Properties of Relaxation

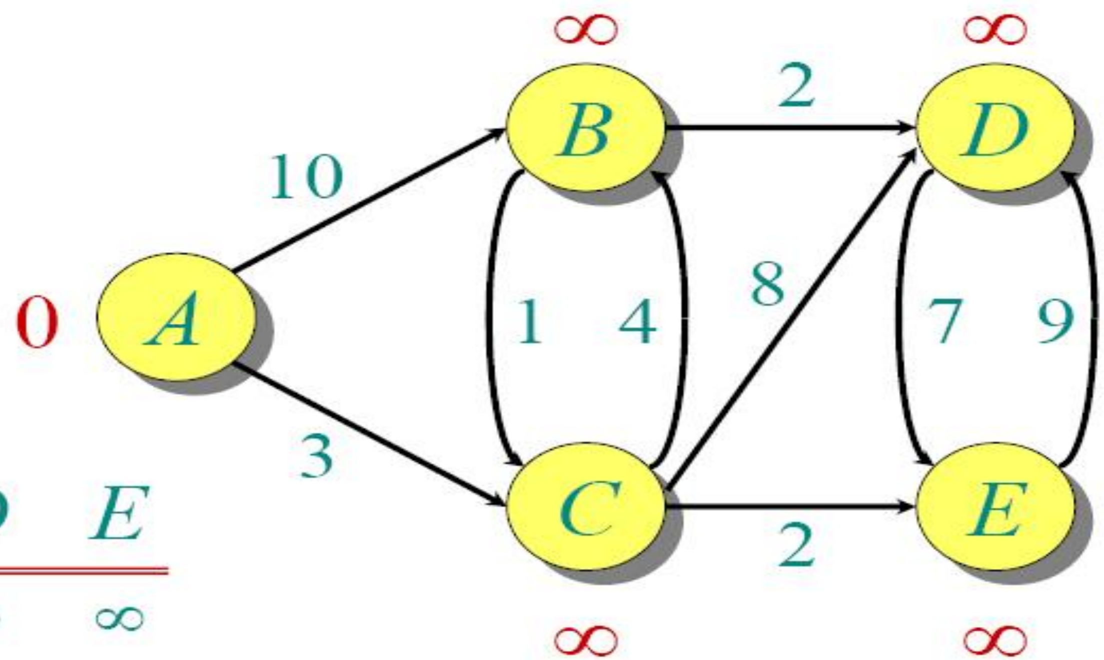
- dist[v], if not ∞, is the length of *some* path from S to v.
- dist[v] either stays the same or decreases with time
- Therefore, if dist[v] = δ(s, v) at any time, this holds thereafter. (δ(s, v) is the weighted path)
- Note that dist[v] ≥ δ(s, v) always
- After i iterations of relaxing on all (u,v), if the shortest path to v has i edges, then dist[v] = δ(s, v).

Dijkstra's algorithm - Pseudocode

```
dist[s] ←0                                  (distance to source vertex is zero)
for  all v ∈ V–{s} do
        dist[v] ←∞                          (set all other distances to infinity)
        P[v] ←Nil                           (Set parent of all vertices)

S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all vertices)


while Q ≠∅ do                               (while the queue is not empty)
    u ← mindistance(Q,dist)                 (select the element of Q with the min. distance)
    S←S∪{u}                                 (add u to list of visited vertices)

    for all v ∈ neighbors[u] do
        if   dist[v] > dist[u] + w(u, v) then          (if new shortest path found)
                dist[v] ←dist[u] + w(u, v)             (set new value of shortest path)
                P[v] ← u
return dist and P
```

Dijkstra Animated Example

Initialize:



Q: A B C D E

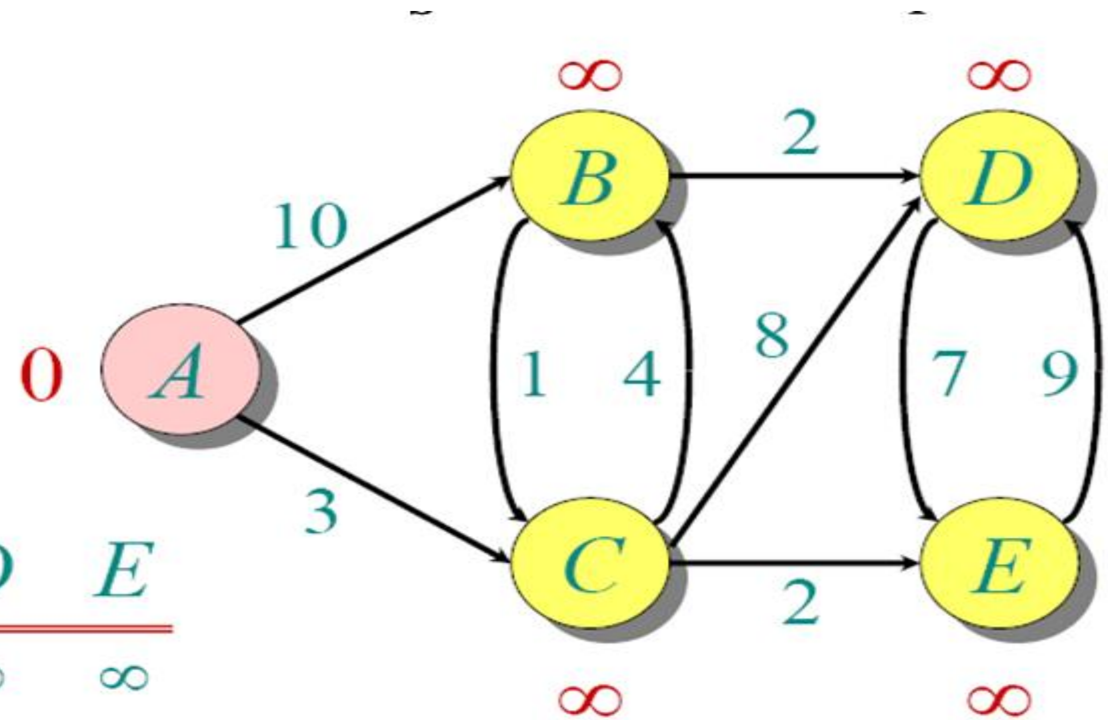
0 ∞ ∞ ∞ ∞

S: {}

P:

A	B	C	D	E
Nil	Nil	Nil	Nil	Nil

Dijkstra Animated Example

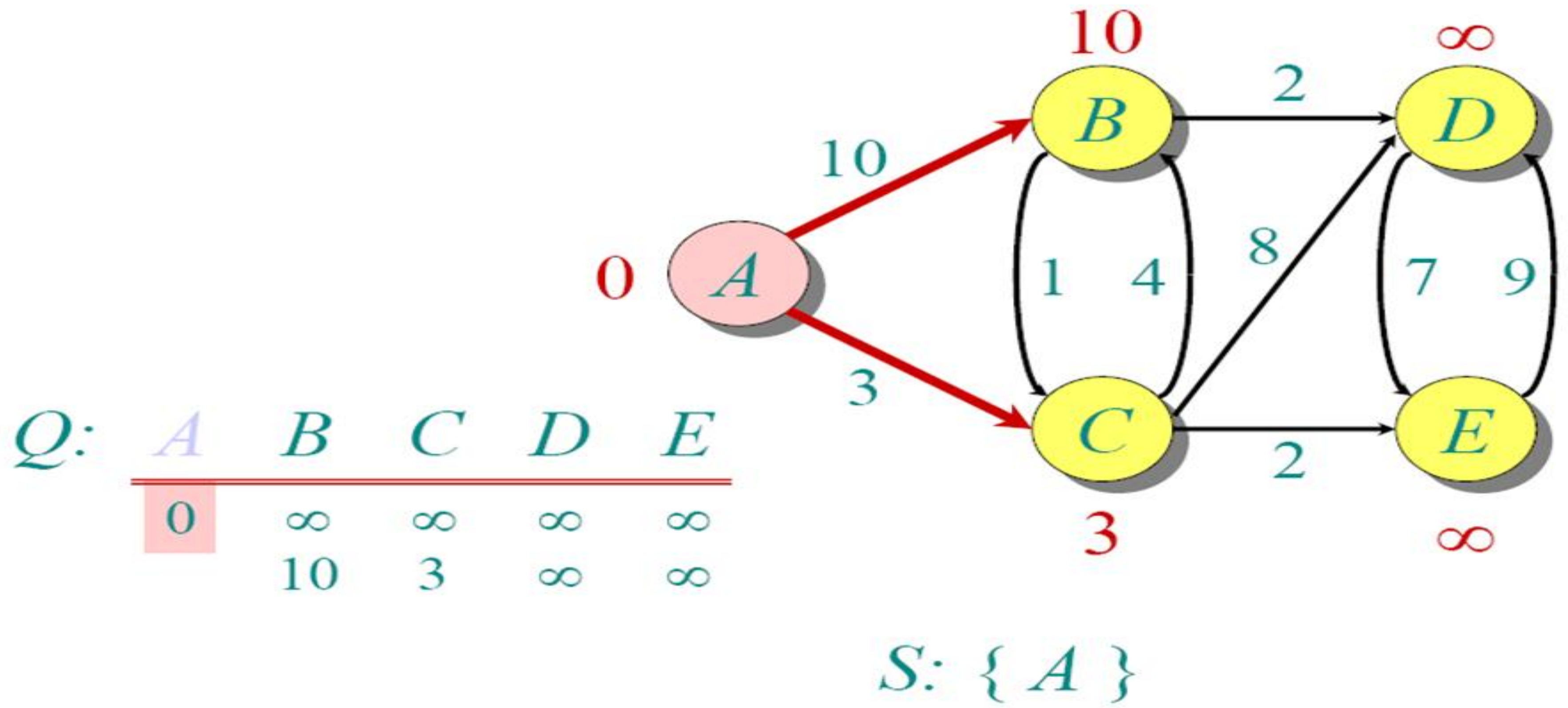


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

P:

A	B	C	D	E
A	Nil	Nil	Nil	Nil

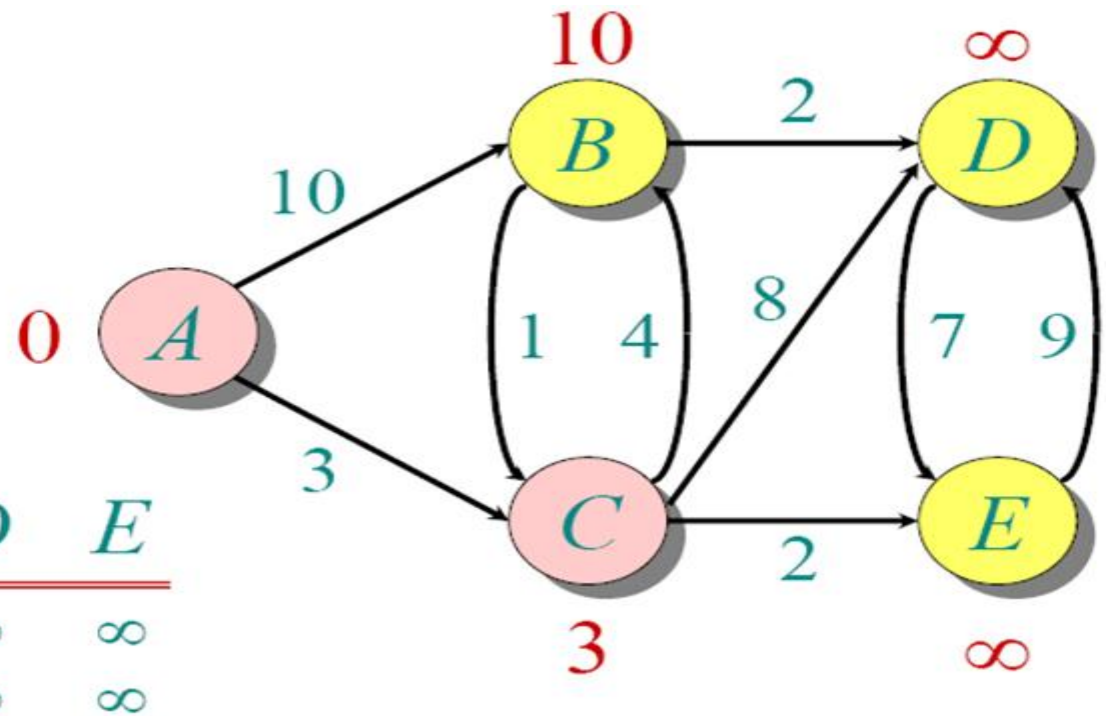
DIJKSTRA ANIMATED EXAMPLE



P:

A	B	C	D	E
A	A	A	Nil	Nil

Dijkstra Animated Example



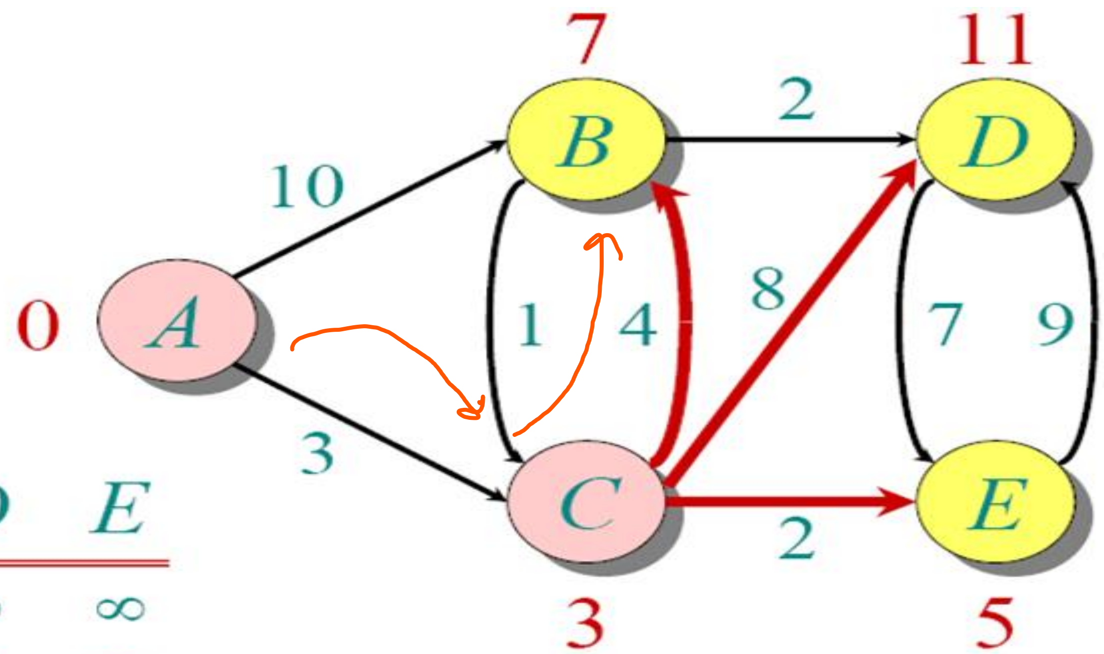
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

P:

A	B	C	D	E
A	A	A	Nil	Nil

Dijkstra Animated Example



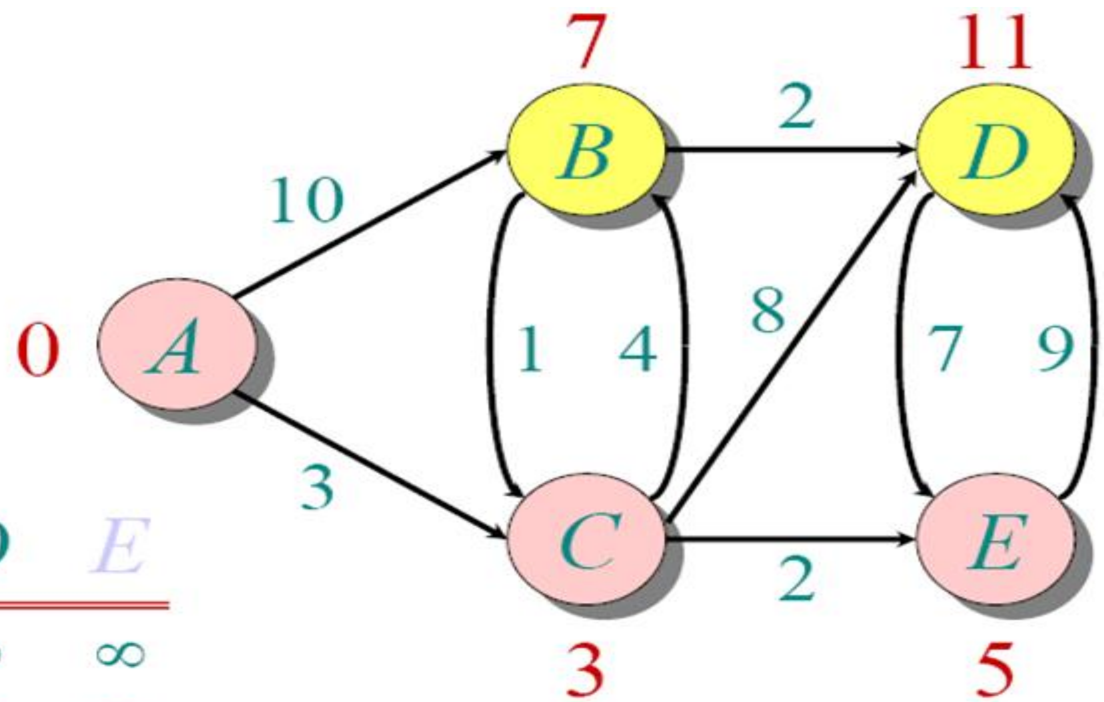
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C }

P:

A	B	C	D	E
A	C	A	C	C

DIJKSTRA ANIMATED EXAMPLE



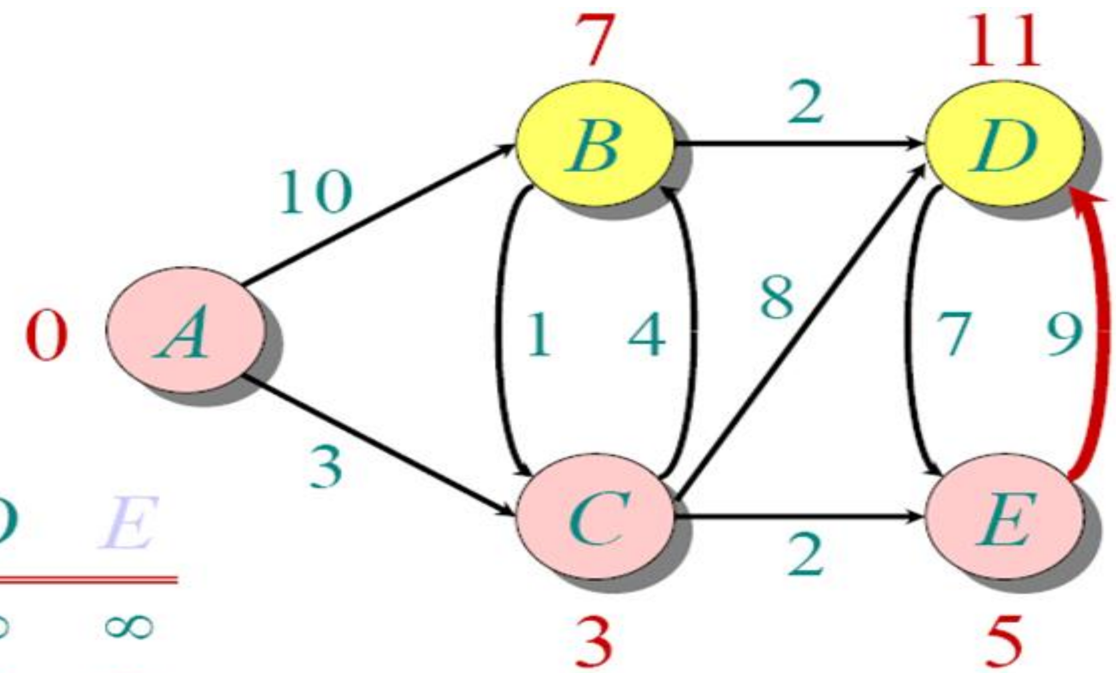
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

P:

A	B	C	D	E
A	C	A	C	C

Dijkstra Animated Example



Q:

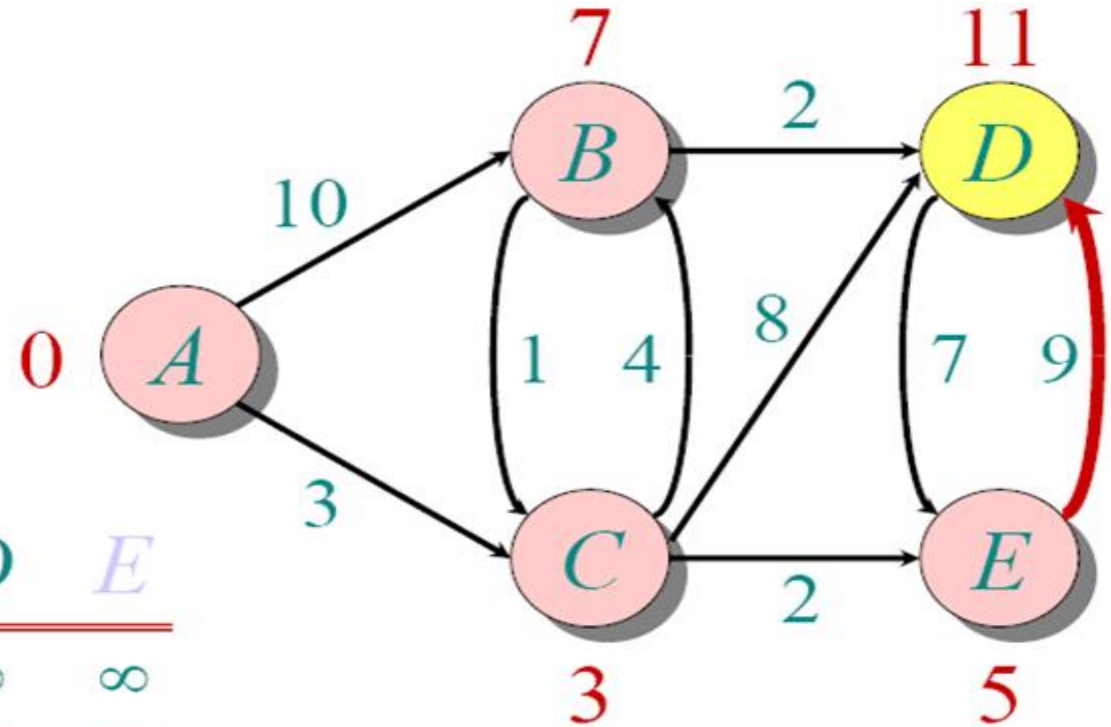
A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	

S: { A, C, E }

P:

A	B	C	D	E
A	C	A	C	C

DIJKSTRA ANIMATED EXAMPLE



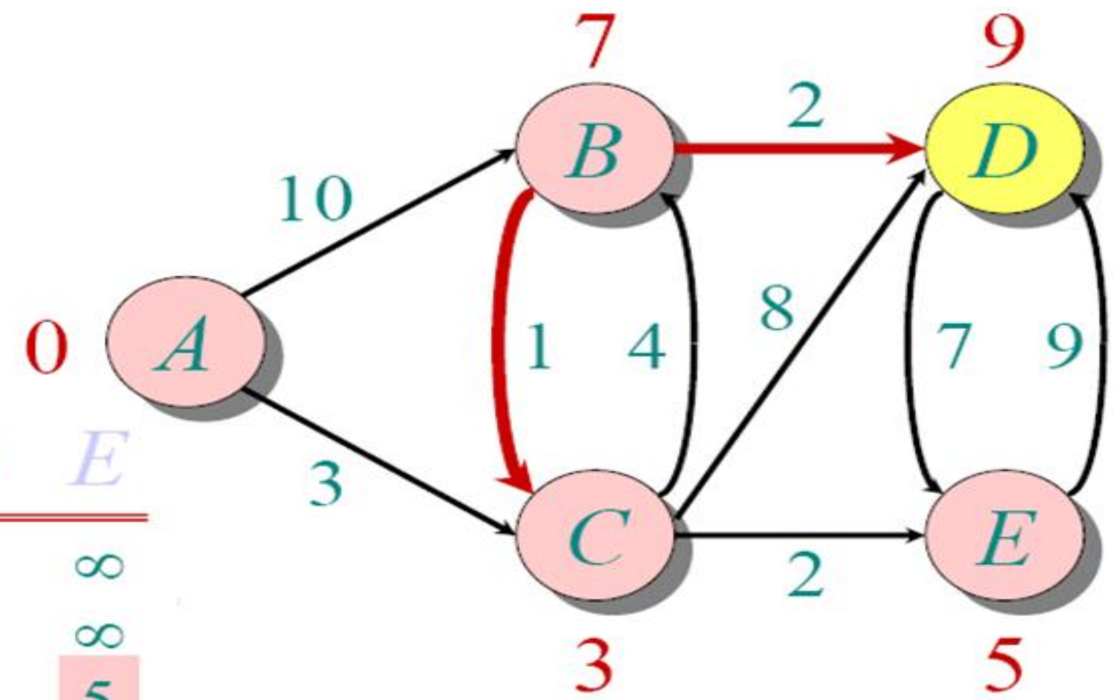
Q: A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	

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

P:

A	B	C	D	E
A	C	A	C	C

DIJKSTRA ANIMATED EXAMPLE



Q:

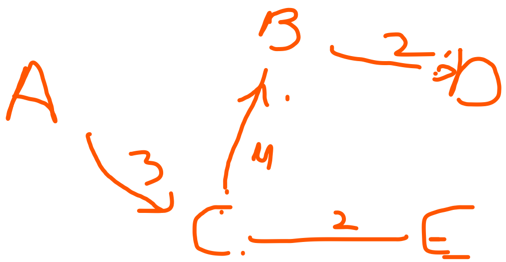
A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	

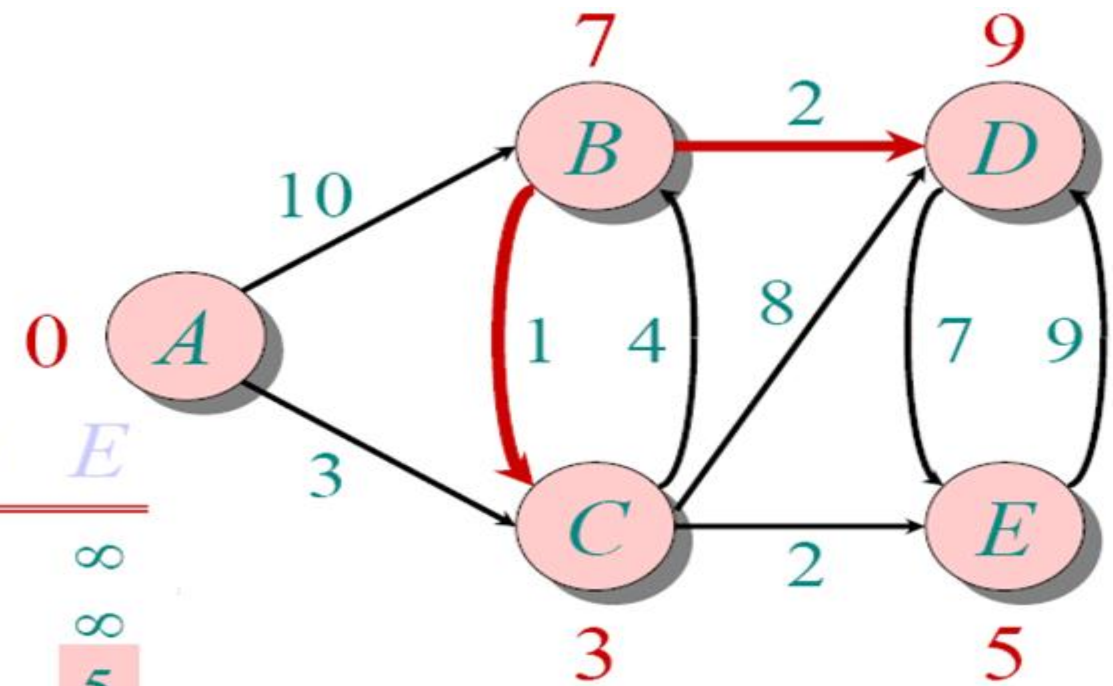
S: { A, C, E, B }

P:

A	B	C	D	E
A	C	A	B	C

DIJKSTRA ANIMATED EXAMPLE





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

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

P:

A	B	C	D	E
A	C	A	B	C

A → E

A → C → E

Implementations and Running Times

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

P:

A	B	C	D	E
A	C	A	B	C

Dijkstra's algorithm - Pseudocode

```
dist[s] ←o
for  all v ∈ V–{s} do
        dist[v] ←∞
        P[v] ←Nil
S←∅
Q←V

while Q ≠∅ do
    u ← mindistance(Q,dist)
    S←S∪{u}
    for all v ∈ neighbors[u] do
        if   dist[v] > dist[u] + w(u, v) then
                dist[v] ←dist[u] + w(u, v)
                P[v] ← u
return dist and P
```

Array	Heaps
O(V)	O(V)
O(V)	O(V)
O(V *V)	O(V * LogV)
O(E)	O(E * LogV)

Implementation? Dense or Sparse??????

	MakeHeap	ExtractMin	Relaxation	Total
Array	$O(\|V\|)$	$O(\|V\|^2)$	$O(\|E\|)$	$O(\|V\|^2)$
Bin heap	$O(\|V\|)$	$O(\|V\| \log \|V\|)$	$O(\|E\| \log \|V\|)$	$O((\|V\|+\|E\|) \log \|V\|)$ $O(\|E\| \log \|V\|)$

D $E = O(V^2)$

- Sparse graph=??? Heaps
- Dense graph=??? Arrays

Dijkstra's Algorithm - Why It Works

- To understand how it works, we'll go over the previous example again. However, we need two mathematical results first:

- **Lemma 1**: Triangle inequality
  If $\delta(u,v)$ is the shortest path length between u and v,
  $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$
- **Lemma 2**:
  The subpath of any shortest path is itself a shortest path.

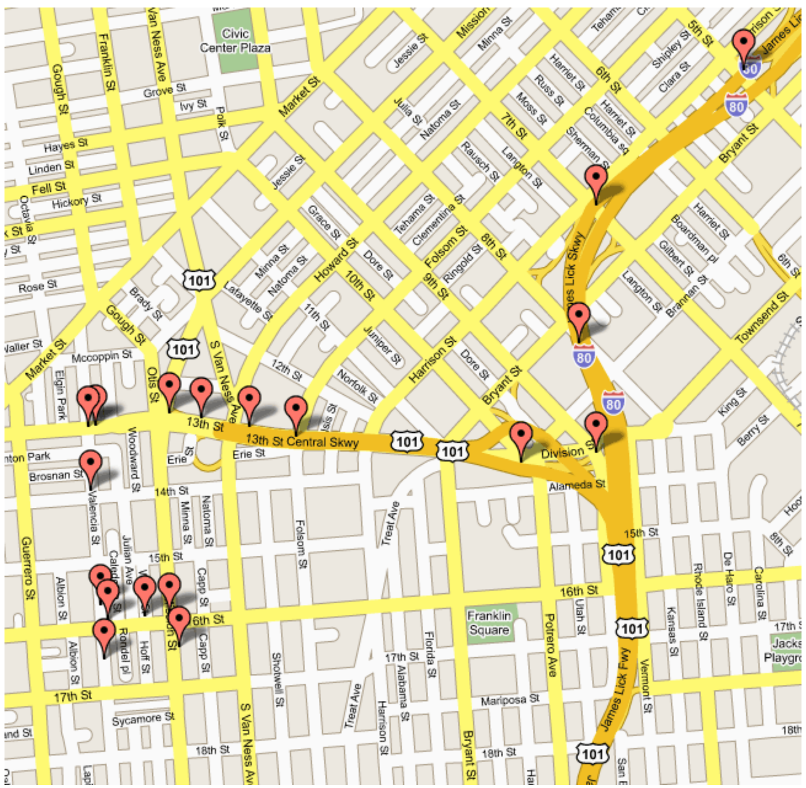
- The key is to understand why we can claim that anytime we put a new vertex in S, we can say that we already know the shortest path to it.
- Now, back to the example…

Dijkstra's Algorithm - Why use it?

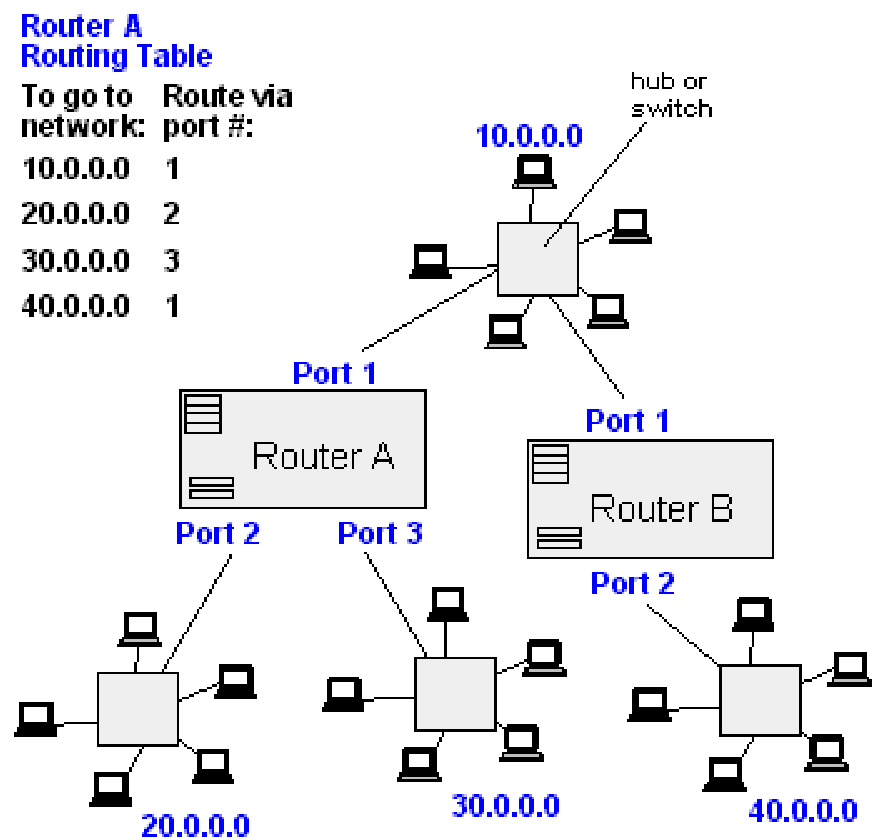
- As mentioned, Dijkstra's algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex u to every vertex using Dijkstra's as it is to calculate the shortest path to some particular vertex v.
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra's algorithm.

APPLICATIONS OF DIJKSTRA'S ALGORITHM

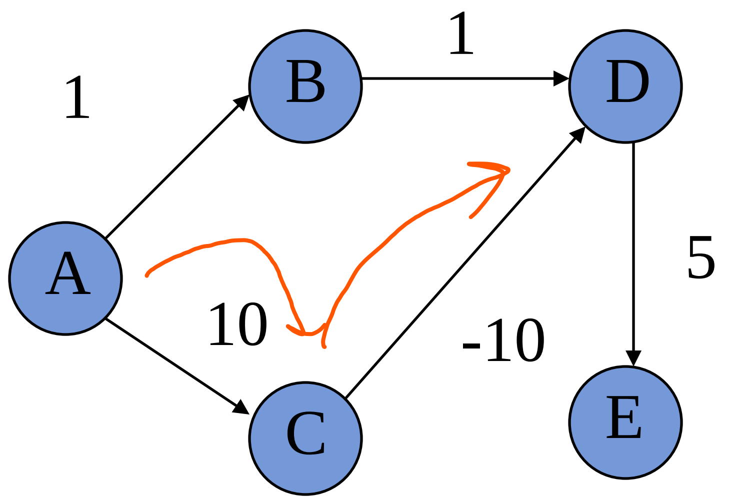
- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.



DIJKSTRA'S ALGORITHM : NEGATIVE WEIGHTS



~~A~~	~~B~~	~~C~~	~~D~~	~~E~~
	∞	∞	∞	∞
0	1	10	∞	∞
		10	2	∞
		10		7
		10		

Wrong

Bounding the distance: Bellman Ford

- For each vertex v, dist[v] is an upper bound on the actual shortest distance
  - start of at ∞
  - only update the value if we find a shorter distance
- An update procedure

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

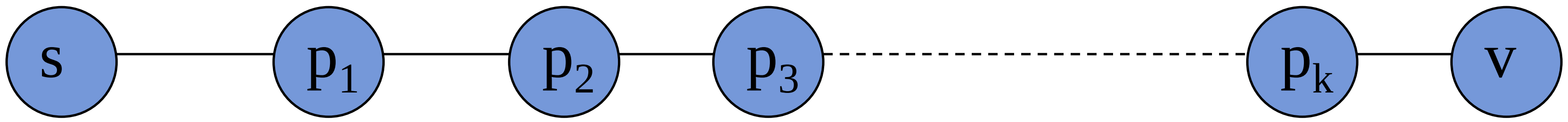
- Can we ever go wrong applying this update rule?
  - We can apply this rule as many times as we want and will never underestimate dist[v]

- When will dist[v] be right?
  - If u is along the shortest path to v and dist[u] is correct

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct

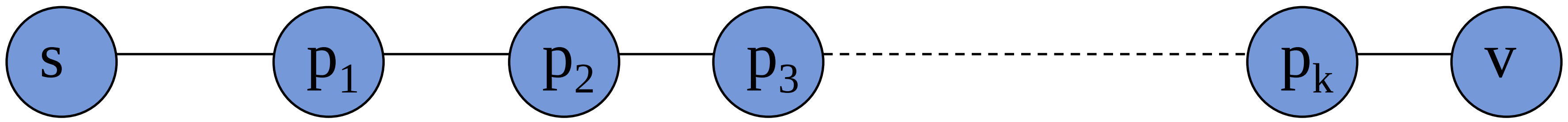
- Consider the shortest path from s to v



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct

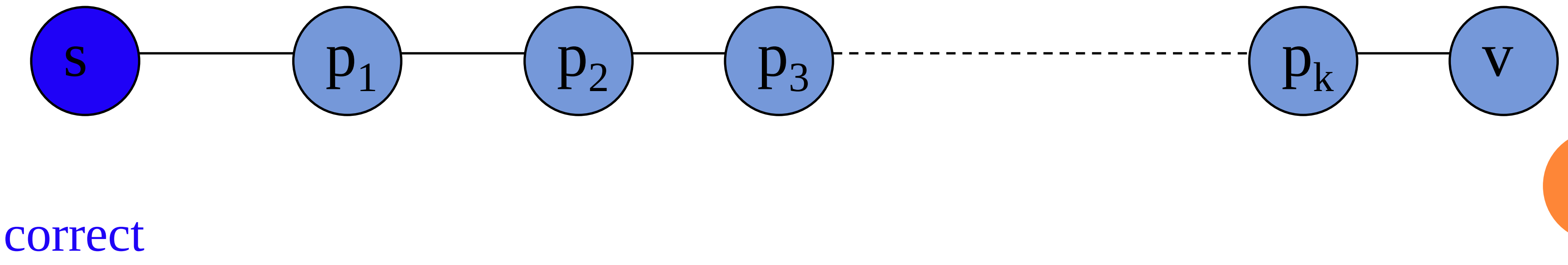
- What happens if we update all of the vertices with the above update?



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct

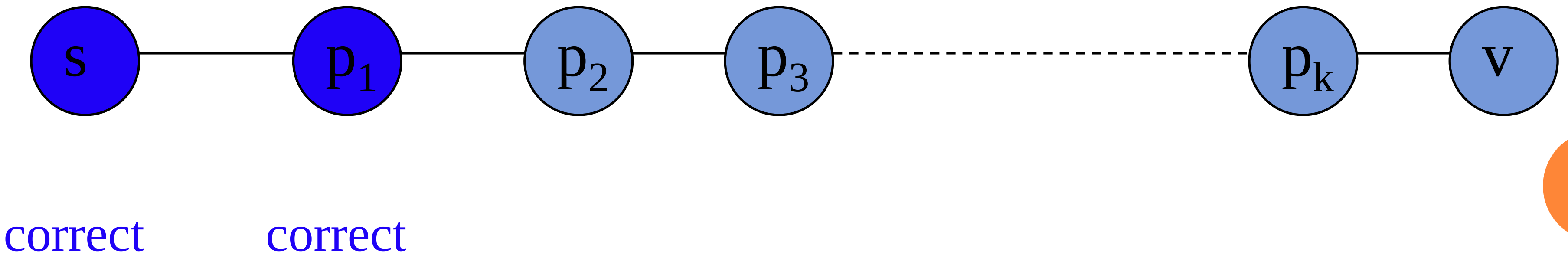
- What happens if we update all of the vertices with the above update?



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct

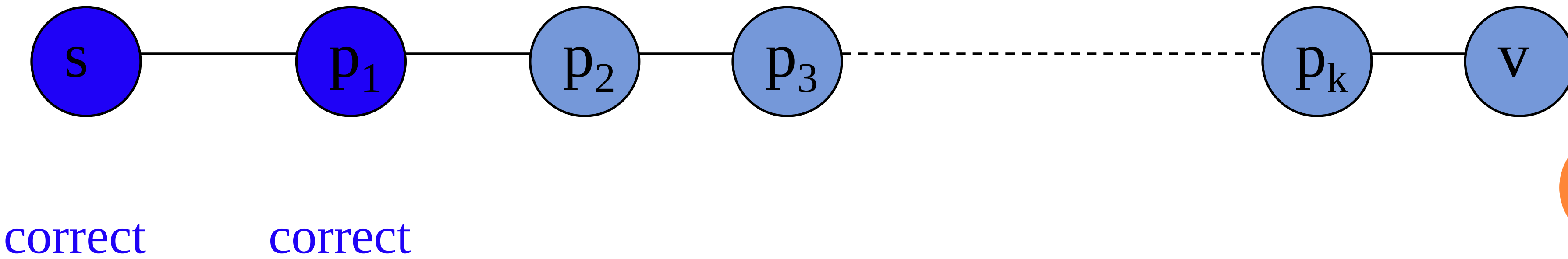
- What happens if we update all of the vertices with the above update?



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct

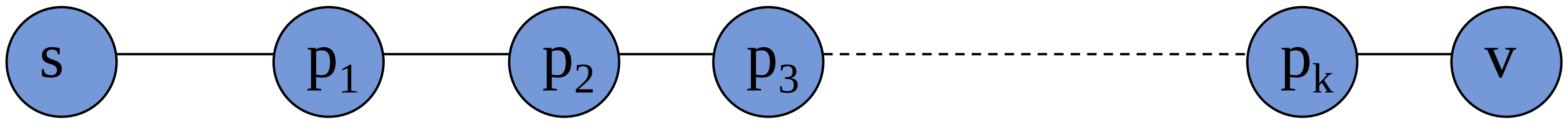
- Does the order that we update the vertices matter?



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

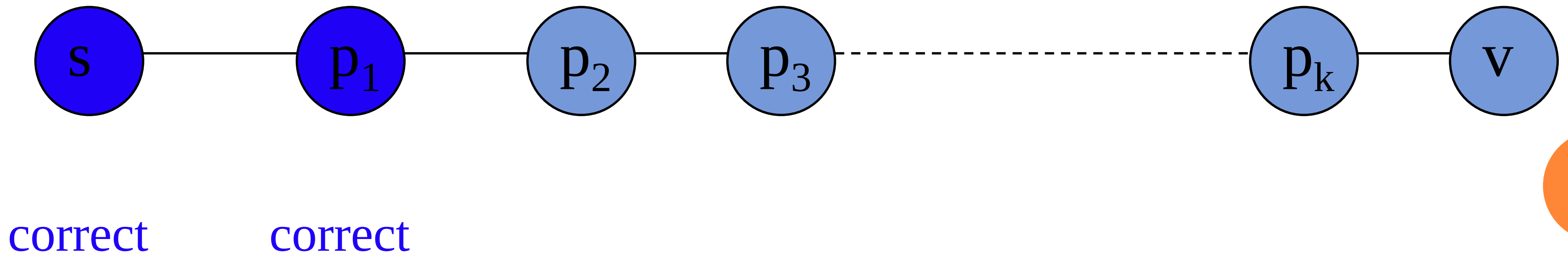
- dist[v] will be right if u is along the shortest path to v and dist[u] is correct

- How many times do we have to do this for vertex p_i to have the correct shortest path from s?



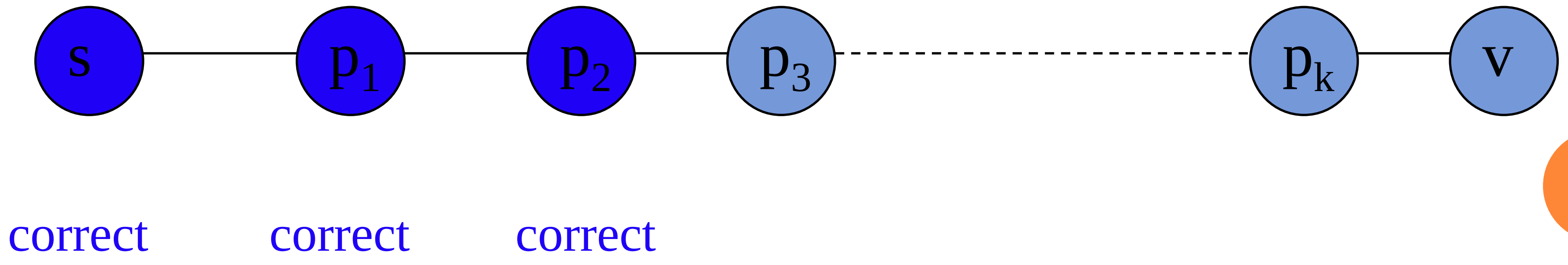
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



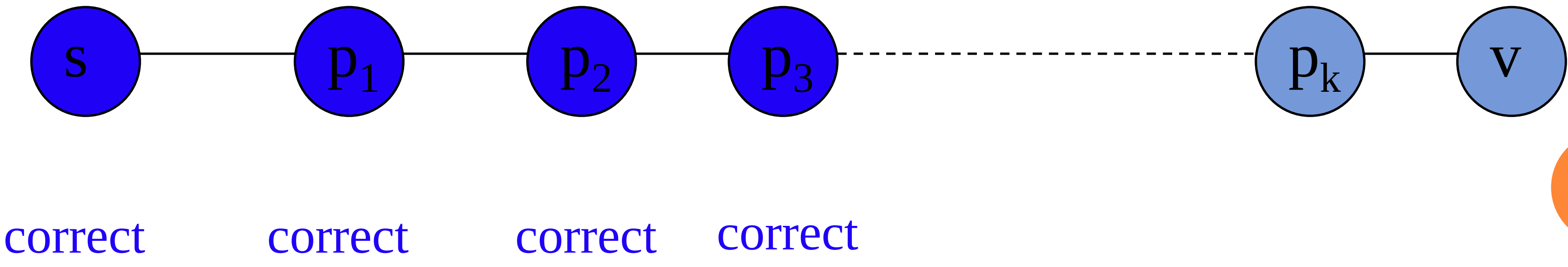
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



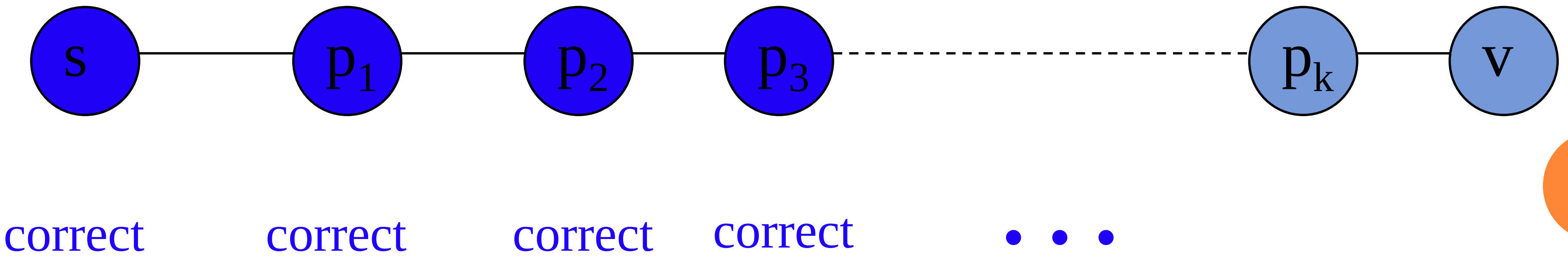
$$dist[v] = \min\{dist[v], dist[u] + w(u, v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



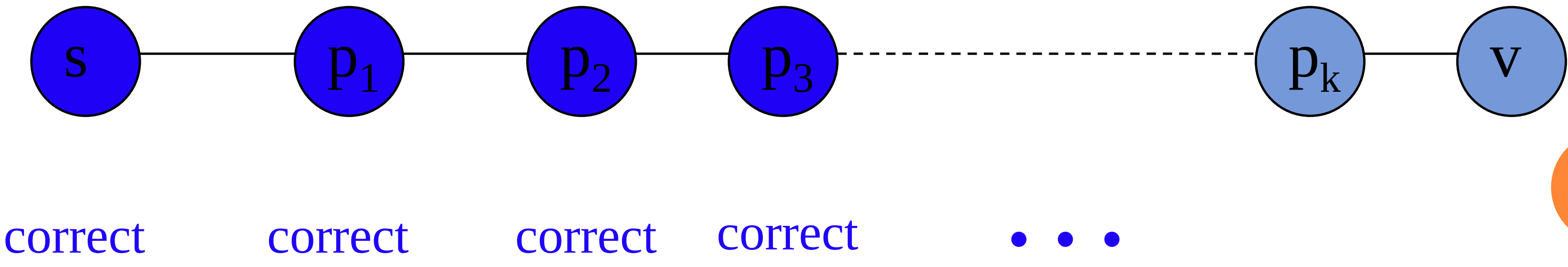
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What is the longest (vetex-wise) the path from s to any node v can be?
  - |V| - 1 edges/vertices



Bellman-Ford algorithm

- Single-source shortest path problem
  - Computes dist(s, v) and P[v] for all $v \in V$
- Allows negative edge weights - can detect negative cycles.
  - Returns TRUE if no negative-weight cycles are reachable from the source s
  - Returns FALSE otherwise $\Rightarrow$ no solution exists

Bellman-Ford algorithm

Bellman-Ford(G, s)

```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

Bellman-Ford algorithm

```
Bellman-Ford(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

Initialize all the distances

BELLMAN-FORD ALGORITHM

```
BELLMAN-FORD(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
for i ← 1 to |V| − 1
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        prev[v] ← u
for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
                return false
```

iterate over all edges/vertices and apply update rule

Bellman-Ford algorithm

Bellman-Ford(G, s)

for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
for $i \leftarrow 1$ **to** $|V| - 1$
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            $prev[v] \leftarrow u$
for all edges $(u, v) \in E$
    if $dist[v] > dist[u] + w(u, v)$
        return $false$

BELLMAN-FORD ALGORITHM

BELLMAN-FORD(G, s)

for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
for $i \leftarrow 1$ **to** $|V| - 1$
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            $prev[v] \leftarrow u$
for all edges $(u, v) \in E$
    if $dist[v] > dist[u] + w(u, v)$
        return $false$

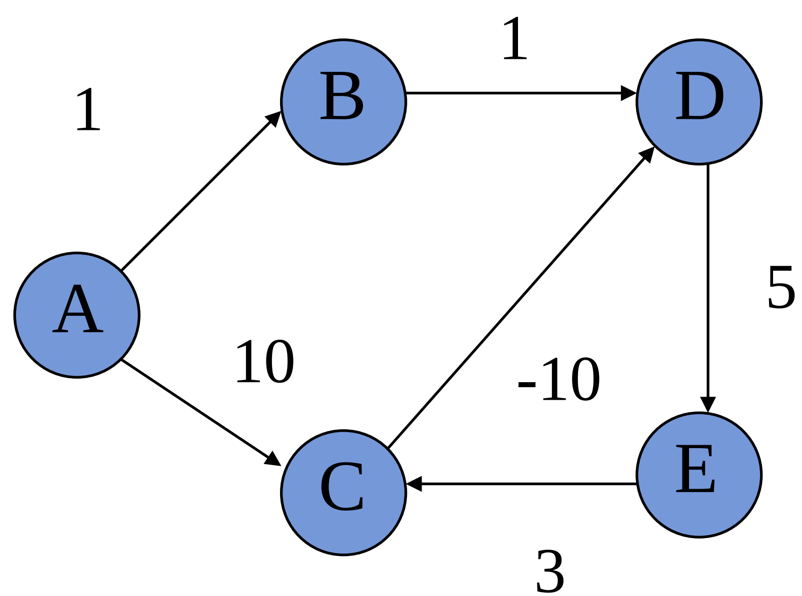
check for negative cycles

Negative cycles

Negative-weight edges may form negative-weight cycles

Negative cycle: A cycle in graph whose total weight is negative

- CDE



→ DECD

$-10 + 3 + 5$

-2

Bellman-Ford algorithm

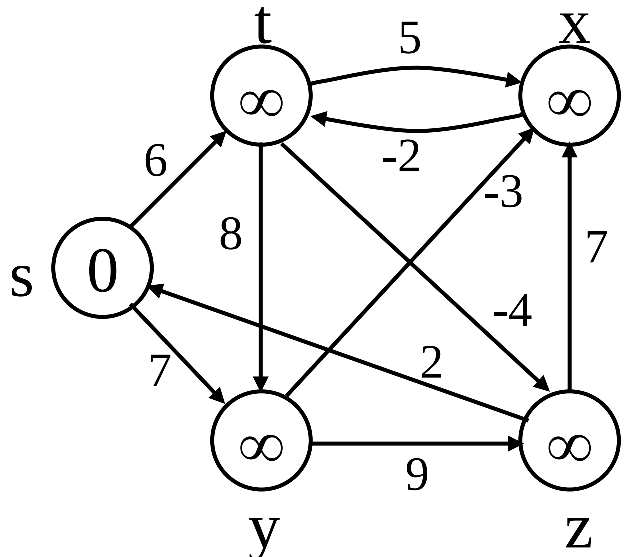
Bellman-Ford(G, s)

for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
for $i \leftarrow 1$ **to** $|V| - 1$
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            $prev[v] \leftarrow u$
for all edges $(u, v) \in E$
    if $dist[v] > dist[u] + w(u, v)$
        return $false$

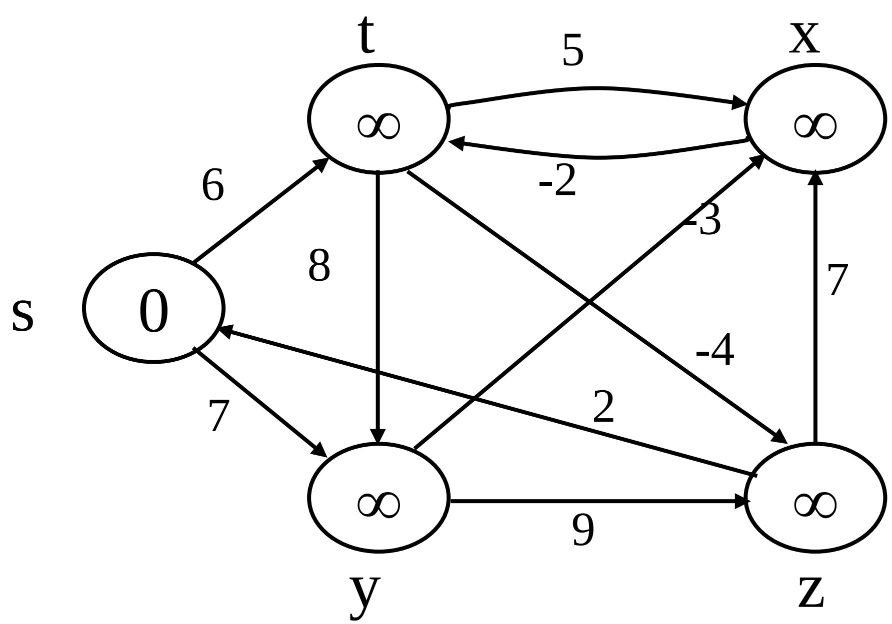
Bellman-Ford Algorithm (cont'd)

- Idea:
  - Each edge is relaxed |V–1| times by making |V-1| passes over the whole edge set.
  - To make sure that each edge is relaxed exactly |V – 1| times, it puts the edges in an unordered list and goes over the list |V – 1| times.

(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)



BELLMAN-FORD(V, E, w, s)



s t x y z
d = 0 ∞ ∞ ∞ ∞
6 7

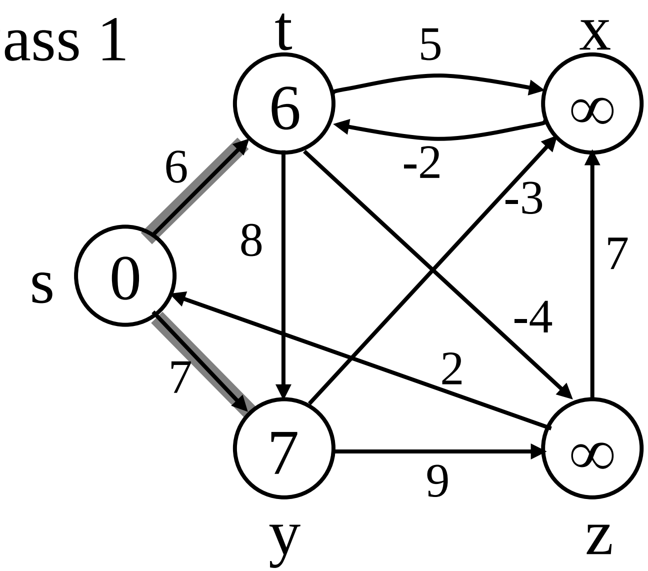
t, x = d[t] + w[t, x]
∞ + 5
∞
d[x] = ∞

E: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

∞ ∞ ∞ ∞ ∞ ∞ ∞ 0 6 7

BELLMAN-FORD(V, E, w, s)

Pass 1



s t x y z

d = 0 6 ~~∞~~ 7 ~~∞~~

~~11~~ 2

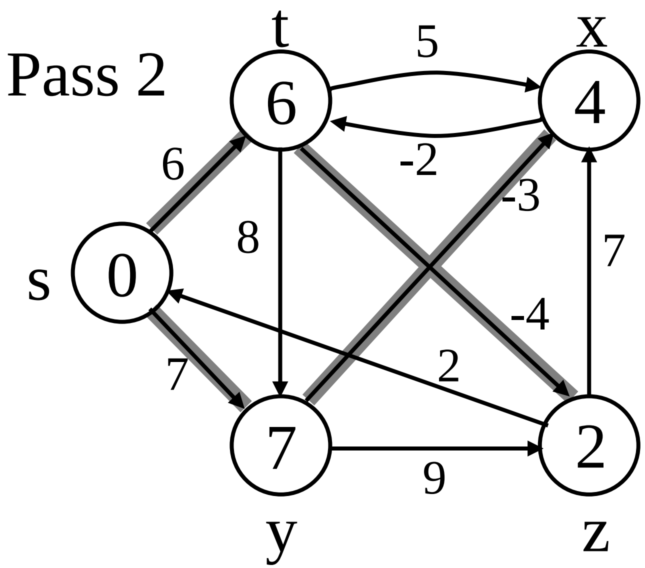
4

$$\begin{cases} d(v) \\ d[u] + w[u, v] \end{cases}$$

E: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

11 14 2 ∞ 4 16 9 6 7

EXAMPLE



s t x y z

d = 0 ~~6~~ 4 7 2

2

d [v] <= d [u] + w [u, v]

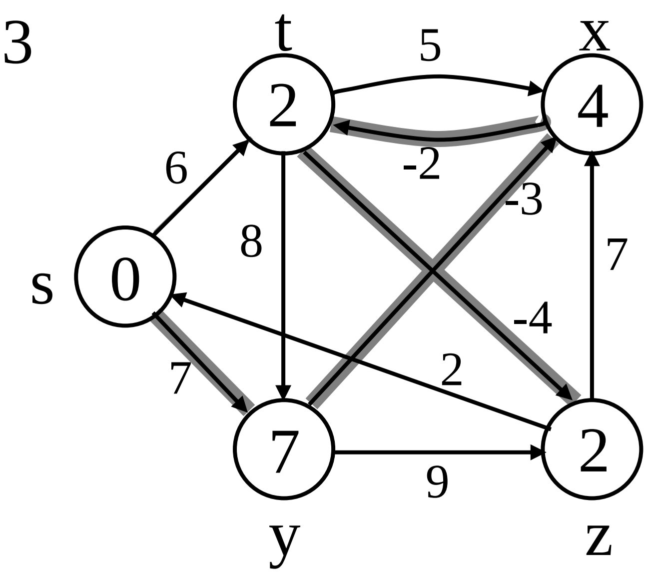
No change

(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

11 14 2 -2

EXAMPLE

Pass 3



Do it

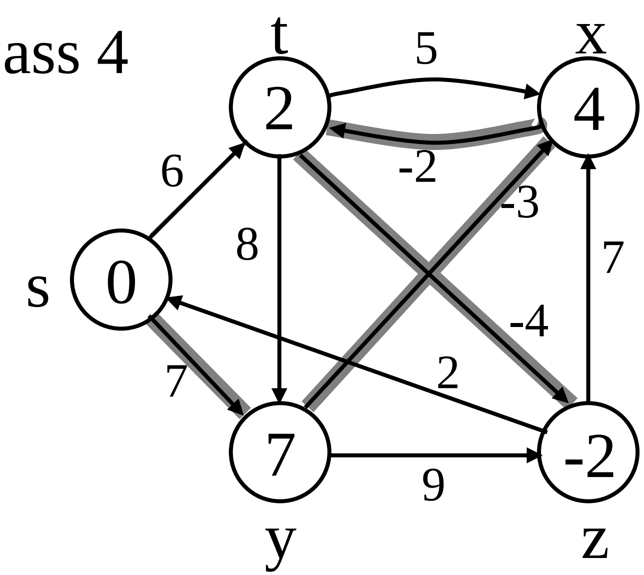
(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

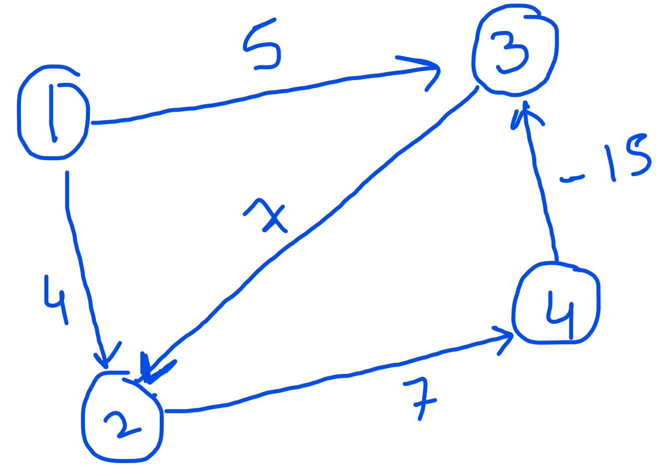
EXAMPLE

Try for 5th

Solve it for 6 iter.

Pass 4





Homework

(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)