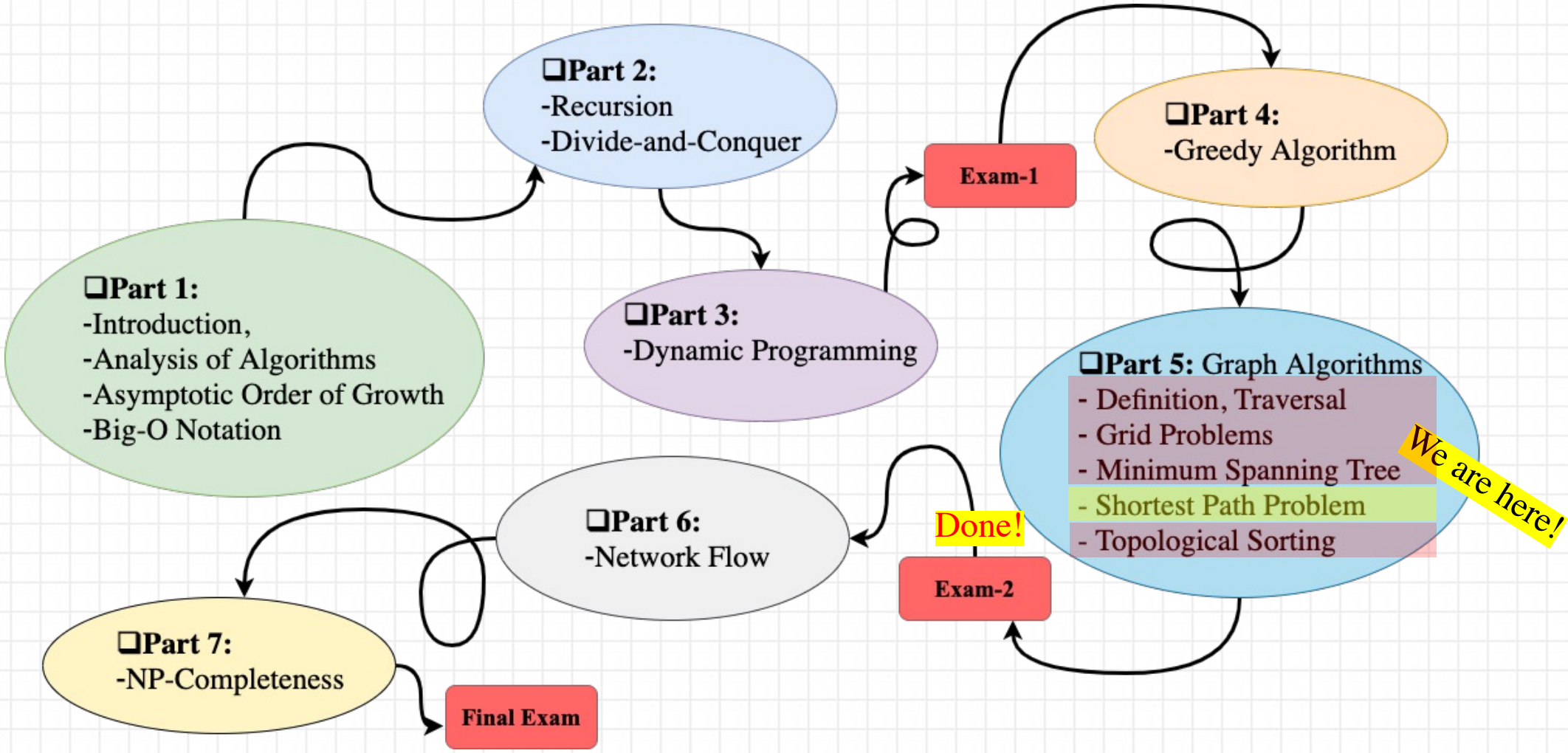
CS-3510:
Design and Analysis of Algorithms

Graph Algorithms:
Shortest Path Problems

Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology
Summer 2022

Roadmap


❑Part 1:
-Introduction,
-Analysis of Algorithms
-Asymptotic Order of Growth
-Big-O Notation
❑Part 2:
-Recursion
-Divide-and-Conquer
❑Part 3:
-Dynamic Programming
Exam-1
❑Part 4:
-Greedy Algorithm
❑Part 5: Graph Algorithms
- Definition, Traversal
- Grid Problems
- Minimum Spanning Tree
- Shortest Path Problem
- Topological Sorting
We are here!
Done!
Exam-2
❑Part 6:
-Network Flow
❑Part 7:
-NP-Completeness
Final Exam

Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components
- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components
- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

Shortest Paths (Weighted Graphs)

- We have already seen that BFS gives the shortest path from a given source node to all other nodes as it traverses the graph, in linear O(|V|+|E|) time. BUT it was limited to unweighted graphs, where the edge weights are the same.

- How can we find the shortest path in weighted graphs?
  - Generalization of BFS to weighted graphs

- Examples
  - Shortest route between two locations on a map
  - Time, cost, penalties, etc.

Shortest Paths (Weighted Graphs)

- Problem description
  - Given graph $G = (V, E)$, and a weight function $w: E \rightarrow \mathbb{R}$
  - Weight of a path $p=[v_0, v_1, \ldots, v_k]$ = $\sum$ edge_weights on path p

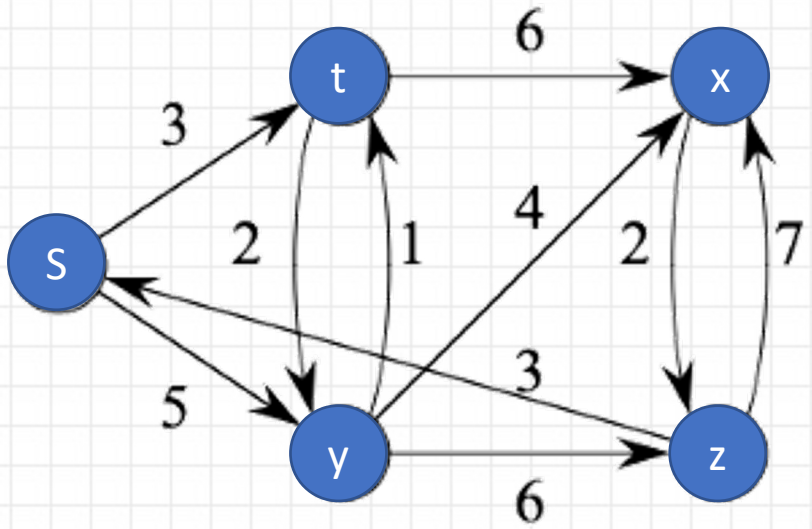
$$= \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

  - Shortest-path weight from $u \rightsquigarrow v$ =

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

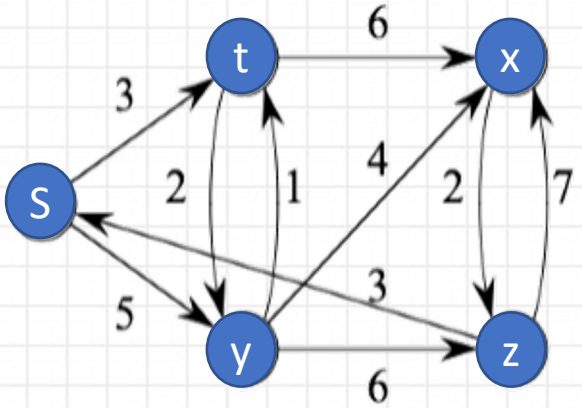
Shortest Paths (Weighted Graphs)

- Example
  - Weighted, directed graph
  - Shortest path from source s to other vertices

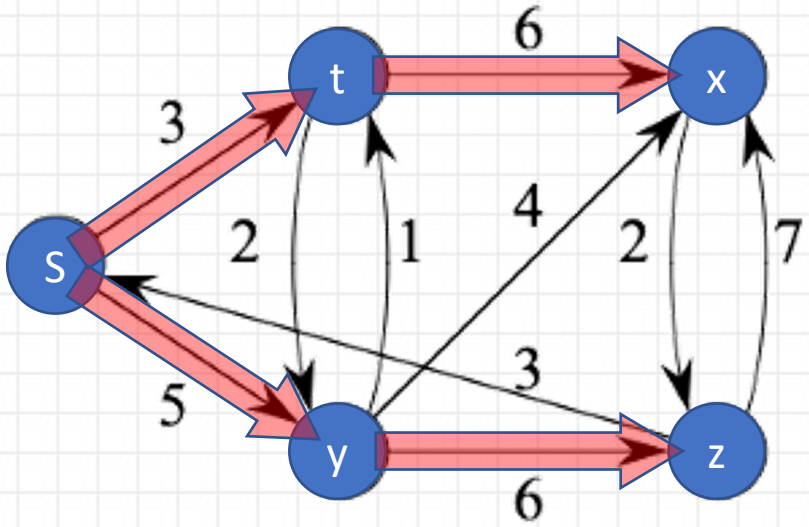


Shortest Paths (Weighted Graphs)

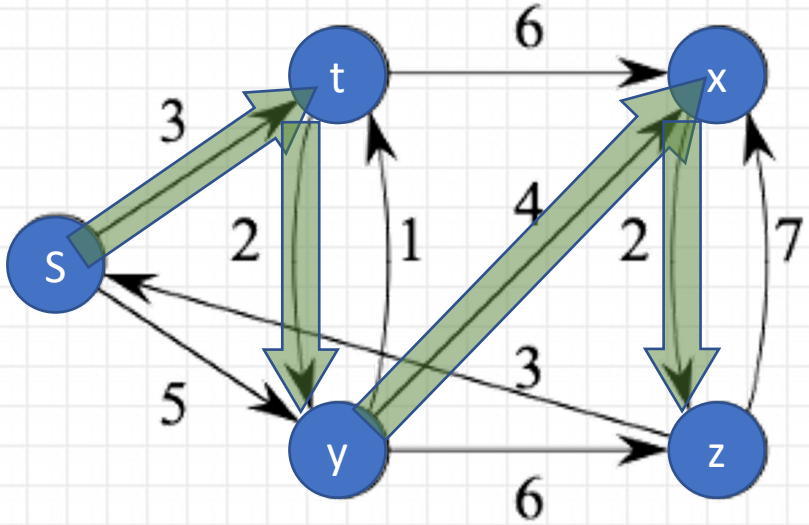
- Example
  - Weighted, directed graph
  - Shortest path from source s to other vertices



Shortest path 1

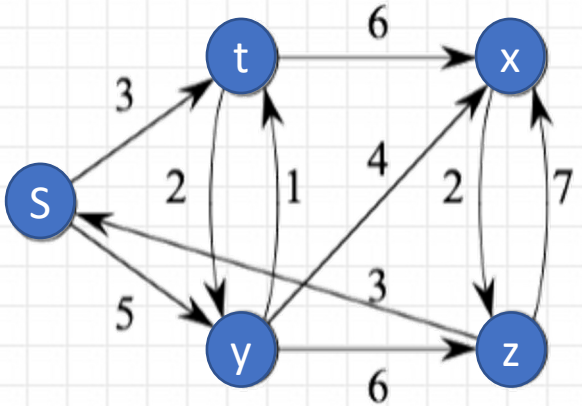


Shortest path 2

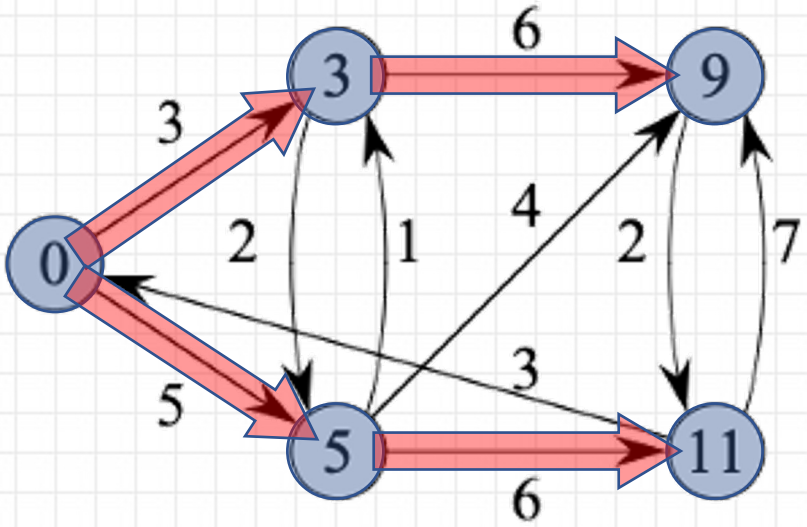


Shortest Paths (Weighted Graphs)

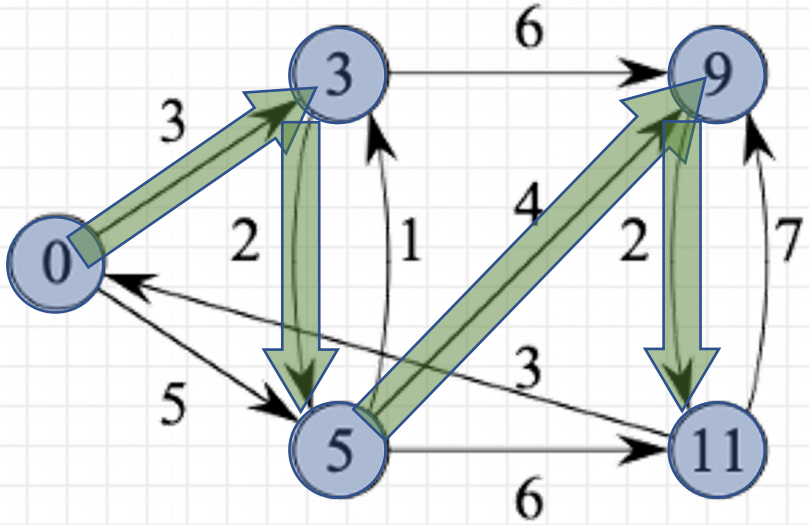
- Example
  - Weighted, directed graph
  - Shortest path from source s to other vertices



Shortest path 1

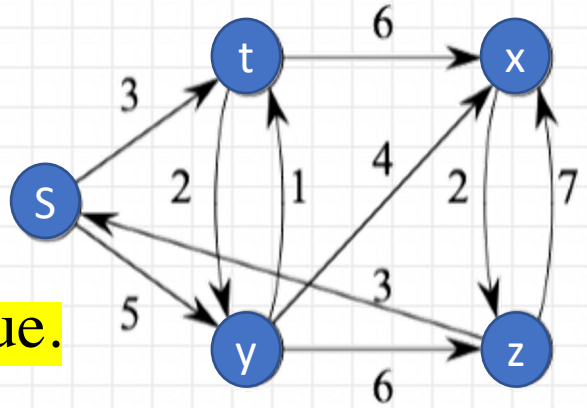


Shortest path 2



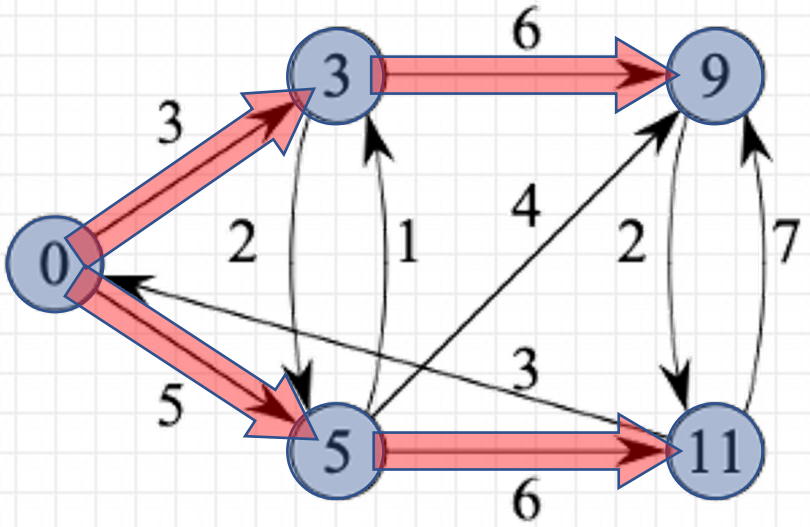
Shortest Paths (Weighted Graphs)

- Example
  - Weighted, directed graph
  - Shortest path from source s to other vertices

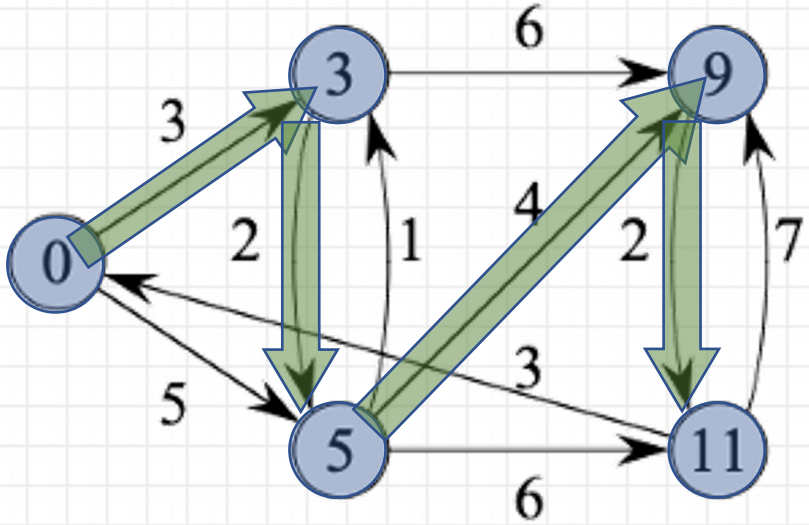


Shortest path does not have to be unique.

Shortest path 1



Shortest path 2



Shortest Paths (Weighted Graphs)

- Variants of shortest path problem:
  1. Single-source: Find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$.
  2. Single-destination: Find shortest paths to a given destination vertex.
  3. Single-pair: Find shortest path from u to v. In the worst case is the same as solving single-source.
  4. All-pairs: Find shortest path from u to v for all $u, v \in V$.

Shortest Paths (Weighted Graphs)

- Variants of shortest path problem:

SSSP

1. Single-source: Find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$.

   Bellman-Ford, Dijkstra

2. Single-destination: Find shortest paths to a given destination vertex.

3. Single-pair: Find shortest path from u to v. In the worst case is the same as solving single-source.

APSP

4. All-pairs: Find shortest path from u to v for all $u, v \in V$.

   Floyd-Warshall

Shortest Paths (Weighted Graphs)

- We start with Single-source shortest path (SSSP) problem first.
  - Bellman-Ford algorithm (Dynamic programming)
  - Dijkstra's algorithm (Greedy)

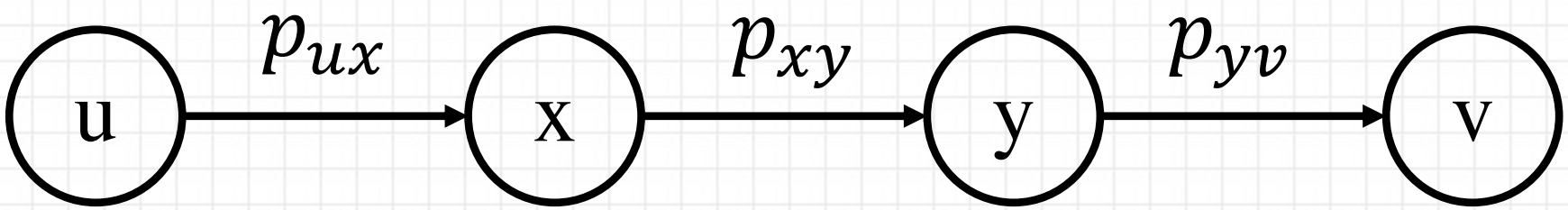
- Then we'll see All-pairs shortest path (APSP) problem
  - Floyd-Warshall

- But before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

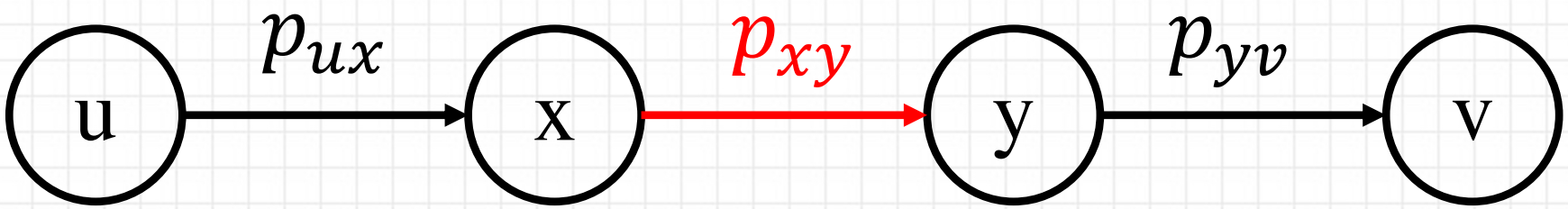
1. Optimal substructure
   - Any sub-path of a shortest path is a shortest path.



Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

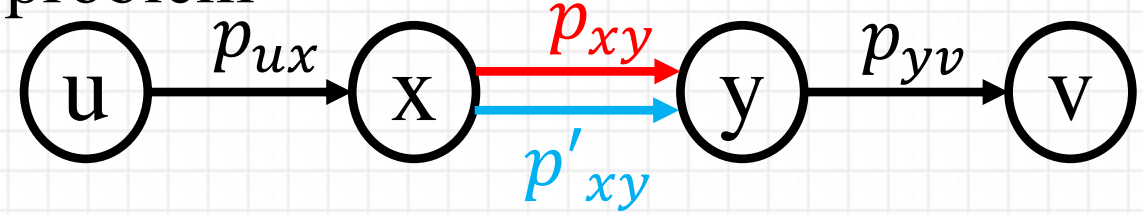
1. Optimal substructure
   - Any sub-path of a shortest path is a shortest path.



$$\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$$

Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem



1. Optimal substructure

- Any sub-path of a shortest path is a shortest path.

$$\delta(u,v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$$

- Proof
  - Suppose p is the shortest path from u to v.
  - For the sake of contradiction assume there is another path, p'_{xy} from x to y, which is shorter than p_{xy}, i.e., $w(p'_{xy}) < w(p_{xy})$. This means we can construct p'_{uv}, using this sub-path:

$$\delta'(u,v) = w(p') = \boxed{w(p_{ux}) + w(p'_{xy}) + w(p_{yv})} < \boxed{w(p_{ux}) + w(p_{xy}) + w(p_{yv})} = w(p)$$

  - Therefore, p wasn't s shortest path → Contradiction.
  - → Each sub-path of a shortest path is a shortest path between the connected nodes.

Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

2. Cycles

Can we have cycles in our shortest path?

Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

2. Cycles

Can we have cycles in our shortest path?

- Positive-weight ⇒ Always a shorter path without the cycle ⇒ omit the cycle.
- Zero-weight: No reason to use them ⇒ we can ignore them in any solution.
- Negative-weight: We cannot have any negative-weight cycle reachable from the source.
  - If we have one, then we can just keep going around it, and get $w(s, v) = -\infty$ for all v on the cycle.
  - Therefore, negative-weight cycle reachable from the source are not allowed.
  - It is okay to have negative-weight cycles which are not reachable from the source.

Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

3. Outputs the SSSP problem

   For each vertex $v \in V$:

   1. Shortest path distance $d[v]$
      - $d[v] = \delta(s,v)$. ($d[v]$ is also known as ***shortest-path estimate)***
      - Initially, $d[v] = \infty$.
      - Reduces as algorithms progress. But always maintain $d[v] \geq \delta(s,v)$.
   2. Shortest path predecessors/ Shortest path tree
      - $\pi[v]$ = predecessor of v on a shortest path from s.
      - If no predecessor, $\pi[v]$ = NIL .
      - π induces a tree known as ***shortest-path tree***.

Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

4. Initialization
   - All the shortest-paths algorithms start with INIT-SINGLE-SOURCE.

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

5. Relaxation
   - **Relaxing an edge** (***u***, v): Updating (improving) the shortest-path estimate for v by going through *u* and taking (*u*,v)

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
   then d[v] ← d[u] + w(u, v)
        π[v] ← u
```

Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

5. Relaxation
   - **Relaxing an edge** (***u***, v): Updating (improving) the shortest-path estimate for v by going through *u* and taking (*u*,v)

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
    then d[v] ← d[u] + w(u, v)
         π[v] ← u
```

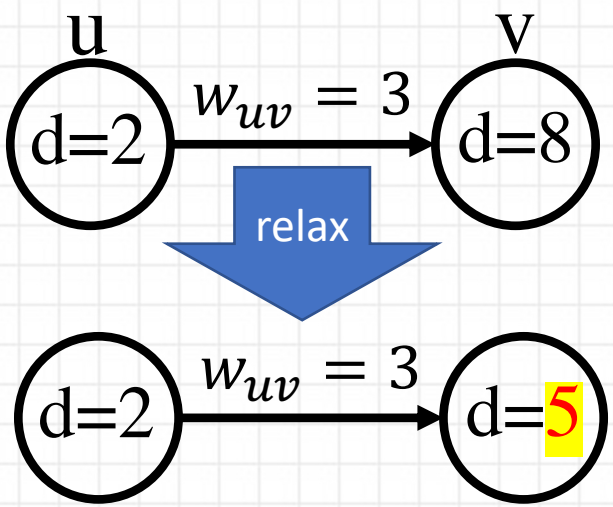
Update only when taking the the current edge reduces the overall path weight sum

Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

5. Relaxation
   - **Relaxing an edge** (***u***, v): Updating (improving) the shortest-path estimate for v by going through *u* and taking (*u*,v)

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
   then d[v] ← d[u] + w(u, v)
        π[v] ← u
```

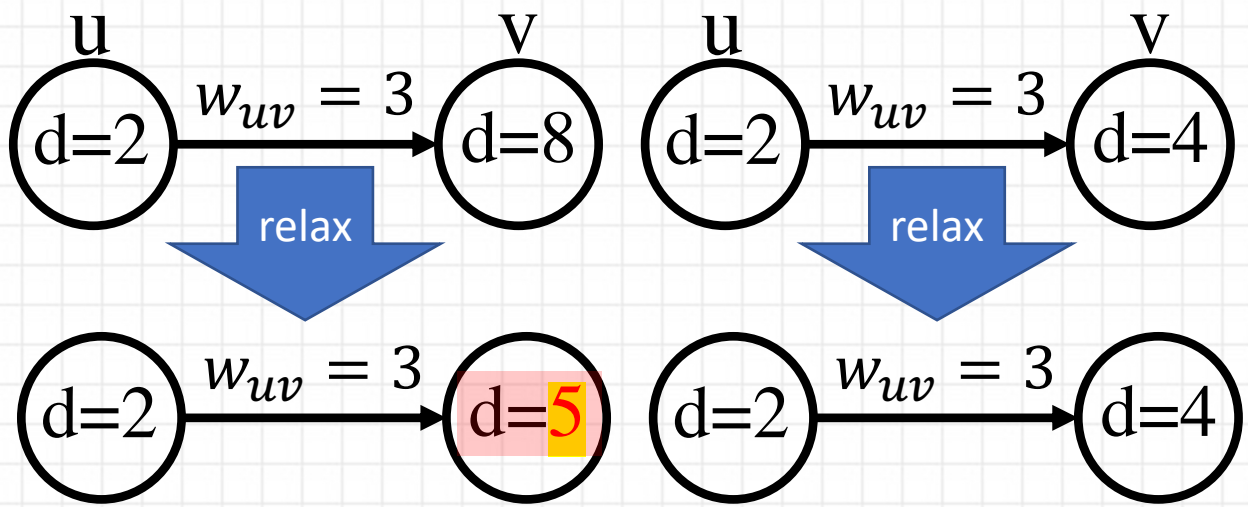


Shortest Paths (Weighted Graphs)

- Before discussing the algorithms, let's see some of the main characteristics of the shortest path problem

5. Relaxation
   - **Relaxing an edge** (***u***, v): Updating (improving) the shortest-path estimate for v by going through *u* and taking (*u*,v)

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
   then d[v] ← d[u] + w(u, v)
        π[v] ← u
```



Shortest Paths (Weighted Graphs)

- The main idea of SSSP algorithms:
  - Calling INIT-SINGLE-SOURCE for initialization.
  - Relaxing edges using Relax method.

- SSSP algorithms are different in terms of the order and number times that the relaxing method is called.

Shortest Paths (Weighted Graphs)

- Shortest path properties
  1. Triangle inequality
  2. Upper-bound property
  3. No-path property
  4. Convergence property
  5. Path relaxation property

Shortest Paths (Weighted Graphs)

- Shortest path properties
  1. Triangle inequality
  2. Upper-bound property
  3. No-path property
  4. Convergence property
  5. Path relaxation property

Shortest Paths (Weighted Graphs)

1. Triangle inequality: For all$(u,v) \in E$, we have $\delta(s,v) \leq \delta(s,u) + w(u,v)$.

- Proof:
  - Weight of shortest path $s \rightsquigarrow v$ is less than or equal to weight of any path $s \rightsquigarrow v$. Path $s \rightsquigarrow u \rightarrow v$ is a path $s \rightsquigarrow v$, and if we use a shortest path $s \rightsquigarrow u$, its weight is $\delta(s, u) + w(u, v)$.

Shortest Paths (Weighted Graphs)

- Shortest path properties
  1. Triangle inequality
  2. Upper-bound property
  3. No-path property
  4. Convergence property
  5. Path relaxation property

Shortest Paths (Weighted Graphs)

2. Upper-bound property: Always have $d[v] \geq \delta(s, v)$ for all v. Once $d[v] = \delta(s, v)$, it never changes.

- Proof:
  - Initially true.
  - Suppose there exists a vertex such that $d[v] < \delta(s, v)$.
  - Without loss of generality, assume v is first vertex for which this happens.
  - Let u be the vertex that causes $d[v]$ to change. Then, $d[v] = d[u] + w(u, v)$.
  - So, $d[v] < \delta(s,v) \leq \delta(s, u) + w(u, v) \leq d[u] + w(u, v) \Rightarrow d[v] < d[u]+w(u,v)$. Contradiction
    (triangle inequality) (v is first violation)
  - Once $d[v]$ reaches $\delta(s,v)$, it never goes lower. It also never goes up, because relaxations only lower shortest-path estimates.

Shortest Paths (Weighted Graphs)

- Shortest path properties
  1. Triangle inequality
  2. Upper-bound property
  3. No-path property
  4. Convergence property
  5. Path relaxation property

Shortest Paths (Weighted Graphs)

3. No-path property: If $\delta(s, v) = \infty$, then $d[v] = \infty$ always.

- Proof:
  - Using upper bound property: $d[v] \geq \delta(s,v) = \infty \Rightarrow d[v] = \infty$.

Shortest Paths (Weighted Graphs)

- Shortest path properties
  1. Triangle inequality
  2. Upper-bound property
  3. No-path property
  4. Convergence property
  5. Path relaxation property

Shortest Paths (Weighted Graphs)

4. Convergence property: If $s \rightsquigarrow u \rightarrow v$ is a shortest path, $d[u] = \delta(s,u)$, and we call RELAX(u,v,w), then $d[v] = \delta(s, v)$ afterward.

- Proof:
  - After relaxation:
    $d[v] \leq d[u]+w(u,v)$ (RELAX code)
    $= \delta(s, u) + w(u, v)$
    $= \delta(s, v)$ (Lemma: optimal substructure)

  - Since $d[v] \geq \delta(s, v)$, must have $d[v] = \delta(s, v)$.

Shortest Paths (Weighted Graphs)

- Shortest path properties
  1. Triangle inequality
  2. Upper-bound property
  3. No-path property
  4. Convergence property
  5. Path relaxation property

Shortest Paths (Weighted Graphs)

5. Path relaxation property: Let p=$[v_0, v_1, \ldots, v_k]$ be a shortest path from $s = v_0$ to v_k. If we relax, *in order*, (v_0, v_1), (v_1, v_2),...,(v_{k-1}, v_k), even intermixed with other relaxations, then $d[v_k] = \delta(s, v_k)$.

- Proof: Induction to show that $d[v_i] = \delta(s, v_i)$ after (v_{i-1}, v_i) is relaxed.
  1. Induction base, $i = 0$: Initially, $d[v_0] = 0 = \delta(s, v_0) = \delta(s, s)$.
  2. Inductive hypothesis: Assume $d[v_{i-1}] = \delta(s, v_{i-1})$
  3. Inductive step: Relax (v_{i-1}, v_i). By convergence property, $d[v_i] = \delta(s, v_i)$ afterward and $d[v_i]$ never changes.

SSSP: Bellman-Ford

- Bellman-Ford
  - Dynamic programming approach
  - Allows negative-weight edges.
  - Computes $d[v]$ and $\pi[v]$ for all $v \in V$.
  - Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

SSSP: Bellman-Ford

- Bellman-Ford
  - Dynamic programming approach
  - Allows negative-weight edges.
  - Computes d[v] and π[v] for all v $\in V$.
  - Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

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

SSSP: Bellman-Ford

- Bellman-Ford
  - Dynamic programming approach
  - Allows negative-weight edges.
  - Computes d[v] and π[v] for all v $\in V$.
  - Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

Relax all edges |V|-1 times

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

SSSP: Bellman-Ford

- Bellman-Ford
  - Dynamic programming approach
  - Allows negative-weight edges.
  - Computes d[v] and π[v] for all v $\in V$.
  - Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

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

Running time?

SSSP: Bellman-Ford

- Bellman-Ford
  - Dynamic programming approach
  - Allows negative-weight edges.
  - Computes d[v] and π[v] for all v $\in V$.
  - Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

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

- Running time: $\Theta(|V||E|)$

Running time?

SSSP: Bellman-Ford

- Bellman-Ford
  - Dynamic programming approach
  - Allows negative-weight edges.
  - Computes d[v] and π[v] for all v $\in V$.
  - Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.
  - Running time: $\Theta(|V||E|)$

  - Proof of correctness using path-relaxation property (CLRS 24.1)

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

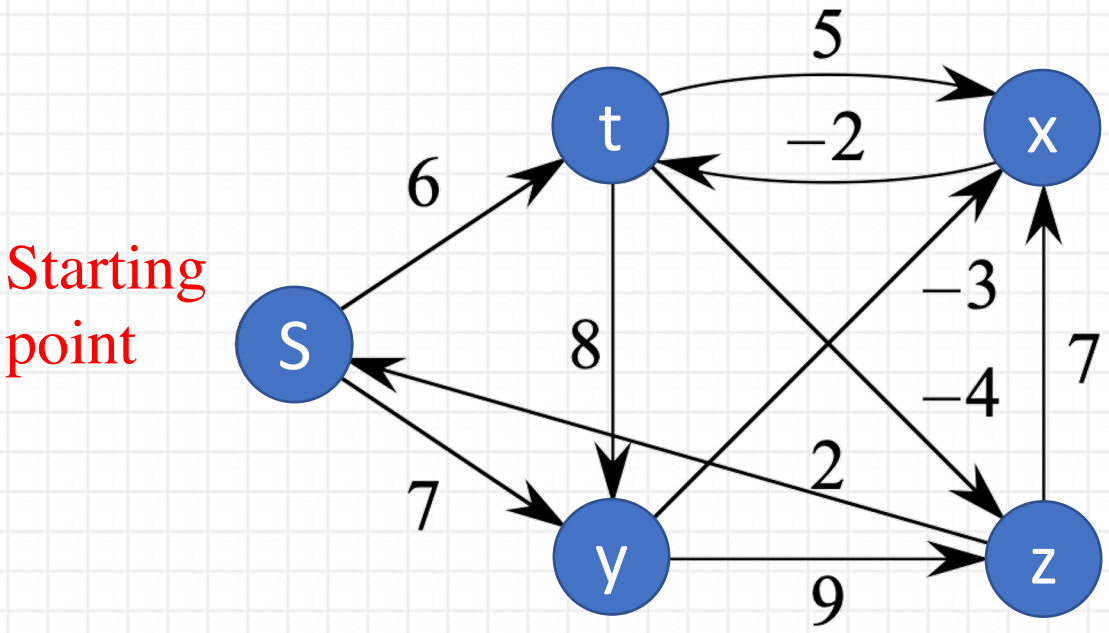
SSSP: Bellman-Ford

- Bellman-Ford
  - So, in short,
  - The algorithm iterates at most |V|-1 times.
  - At each iteration, it updates (relaxes) along all edges.

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

SSSP: Bellman-Ford

- Example
  - 5 vertices → 4 iterations




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

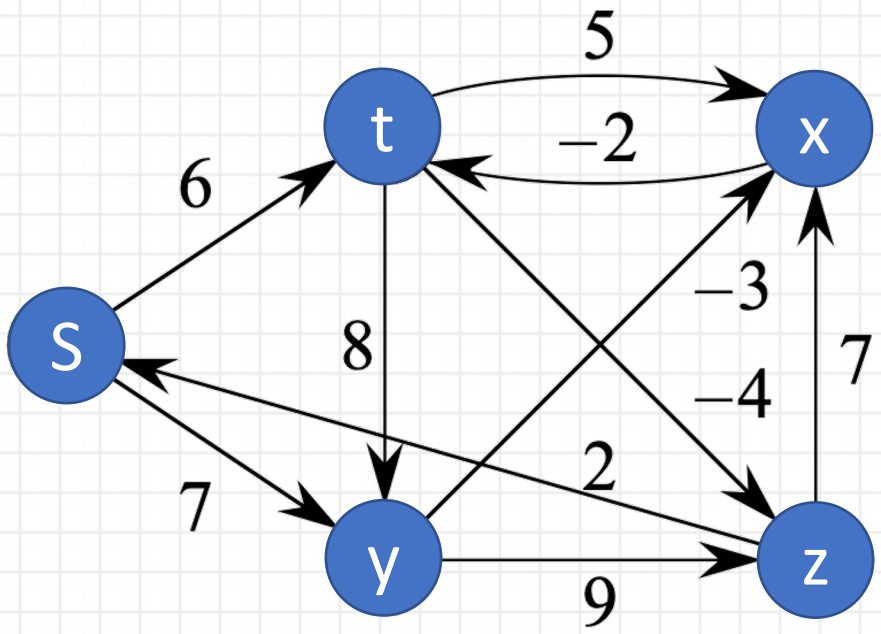
SSSP: Bellman-Ford

- Example
  - 5 vertices → 4 iterations

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

Parents (path)

v	π(v)
s	
t	
y	
x	
z	



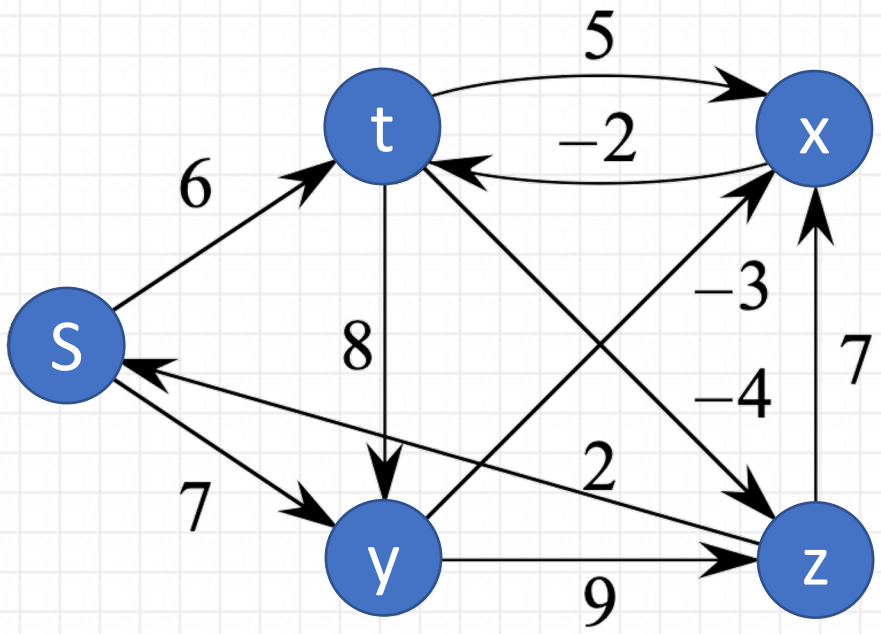
i	s	t	y	x	z
0					
1					
2					
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

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



Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

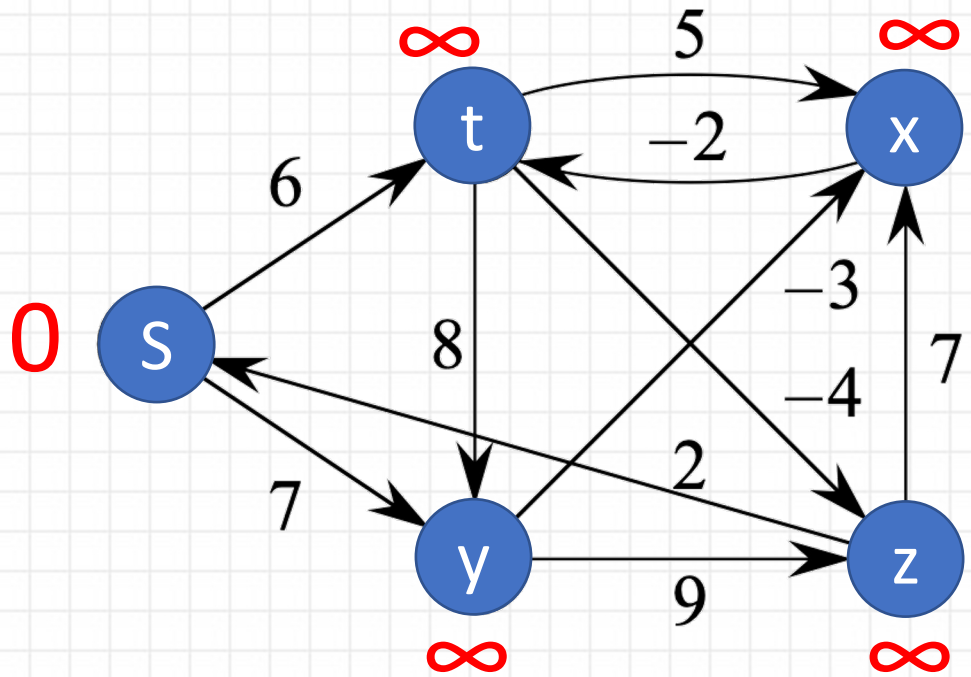
i	s	t	y	x	z
0	0	∞	∞	∞	∞
1					
2					
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

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



Parents (path)

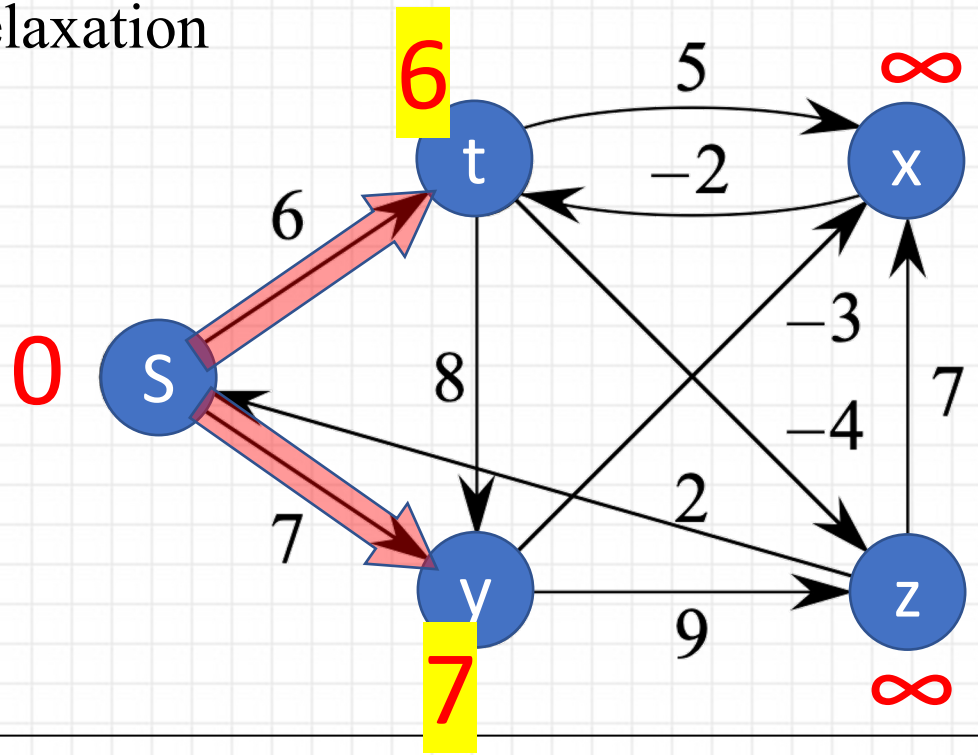
v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1					
2					
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=1



Parents (path)

v	π(v)
s	∅
t	s
y	s
x	∅
z	∅

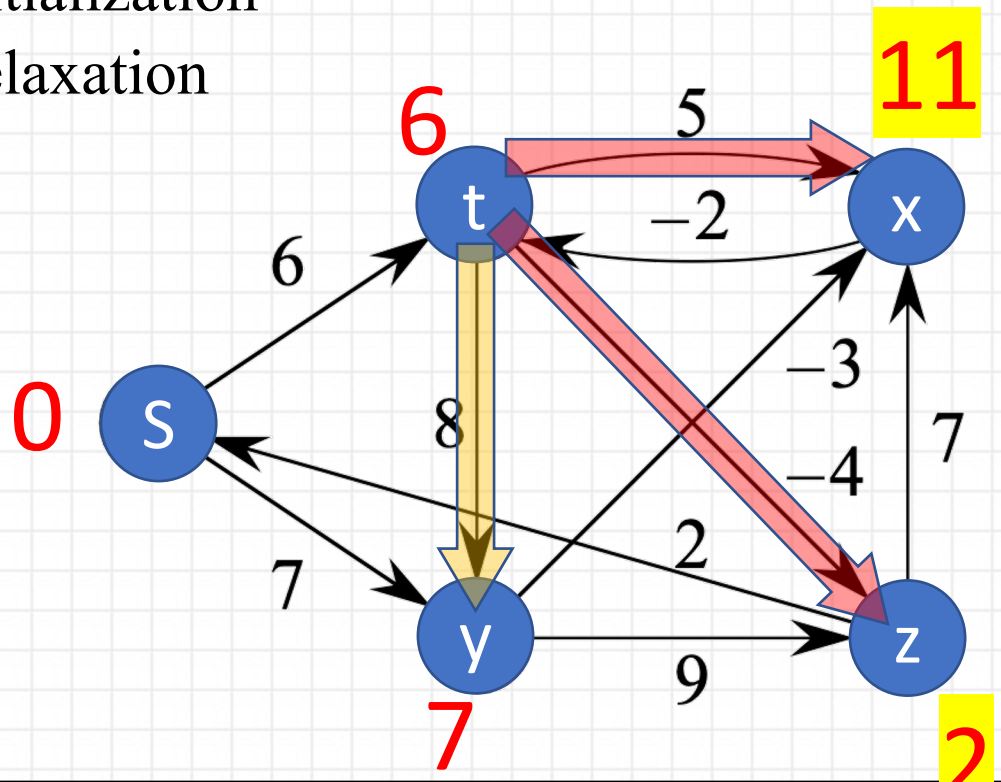
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7		
2					
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=1



Parents (path)

v	π(v)
s	∅
t	**s**
y	**s**
x	**t**
z	**t**

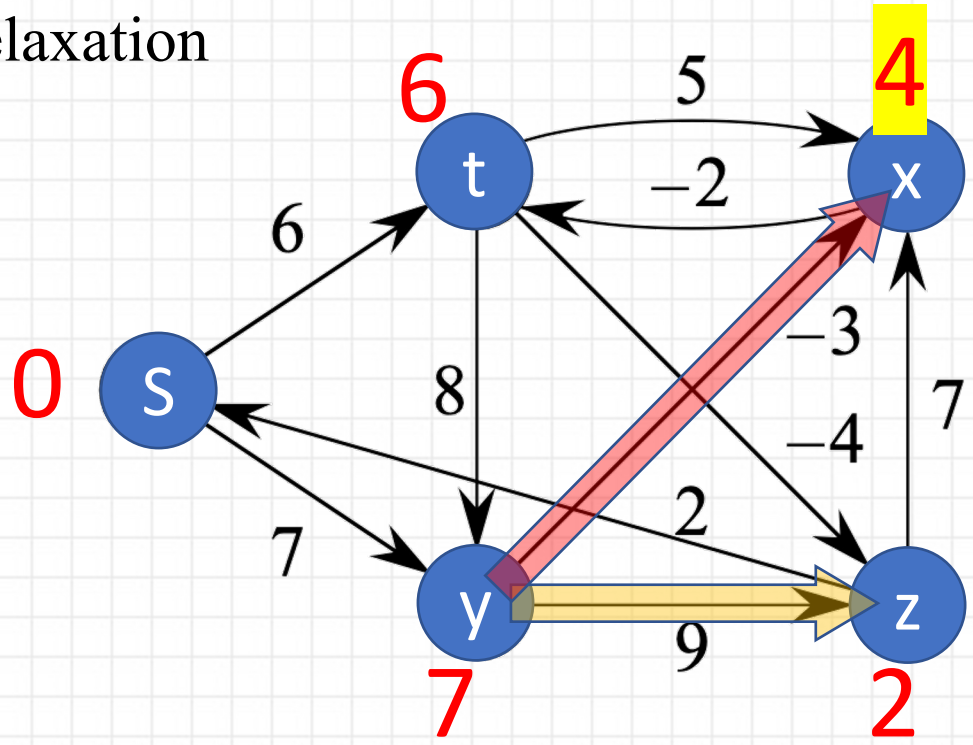
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7	11	2
2					
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=1



Parents (path)

v	π(v)
s	∅
t	s
y	s
x	y
z	t

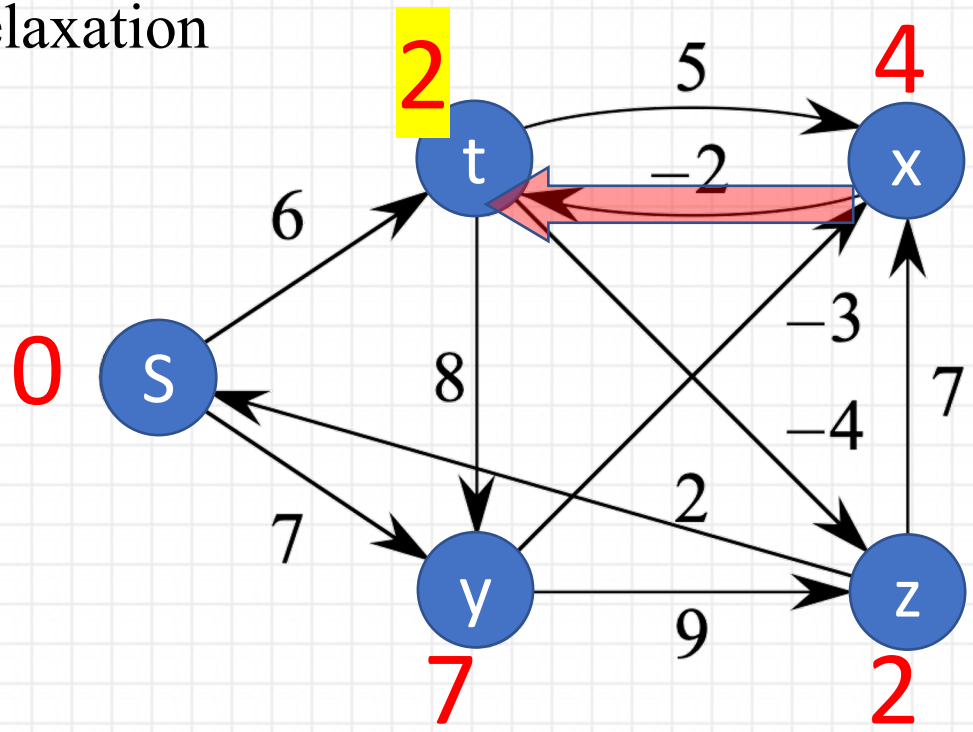
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	6	7	4	2
2					
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=1



Parents (path)

v	π(v)
s	∅
t	x
y	s
x	y
z	t

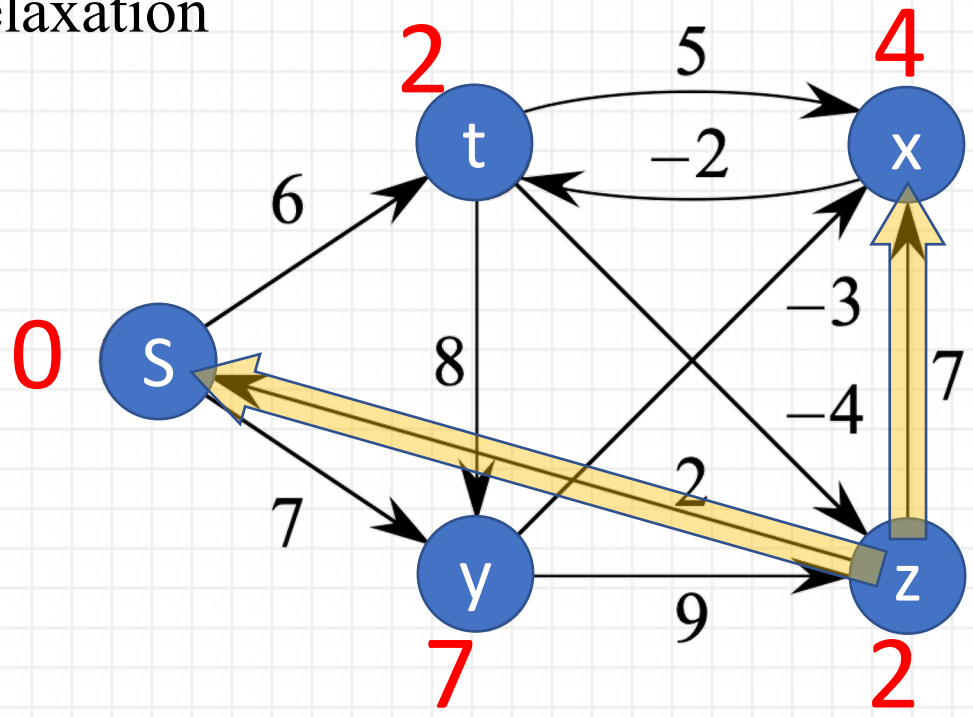
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2					
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=1



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

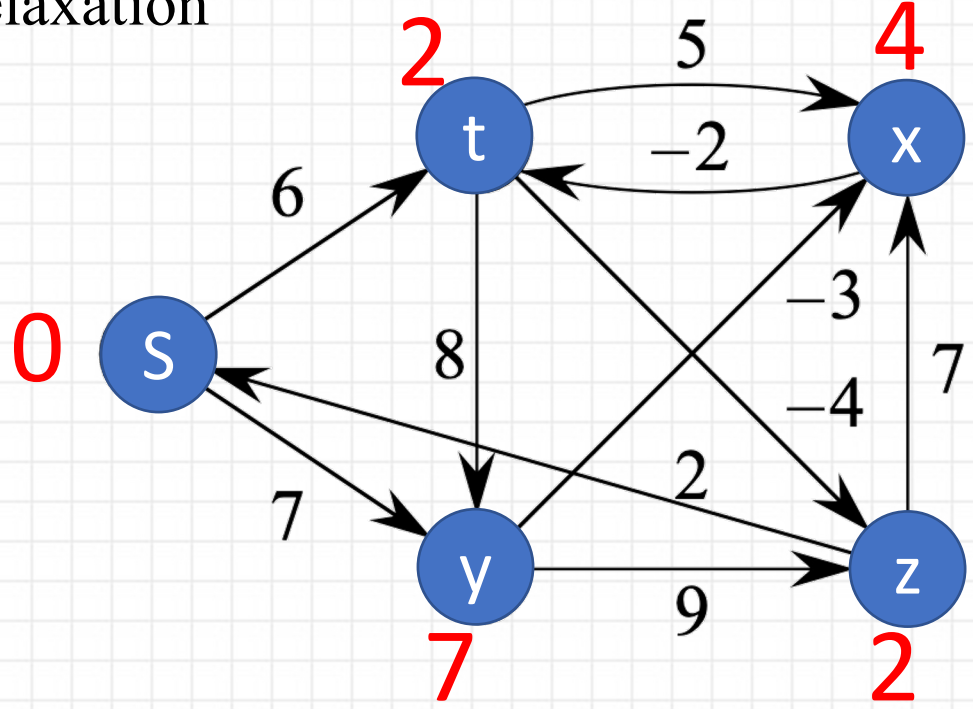
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2					
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=2



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

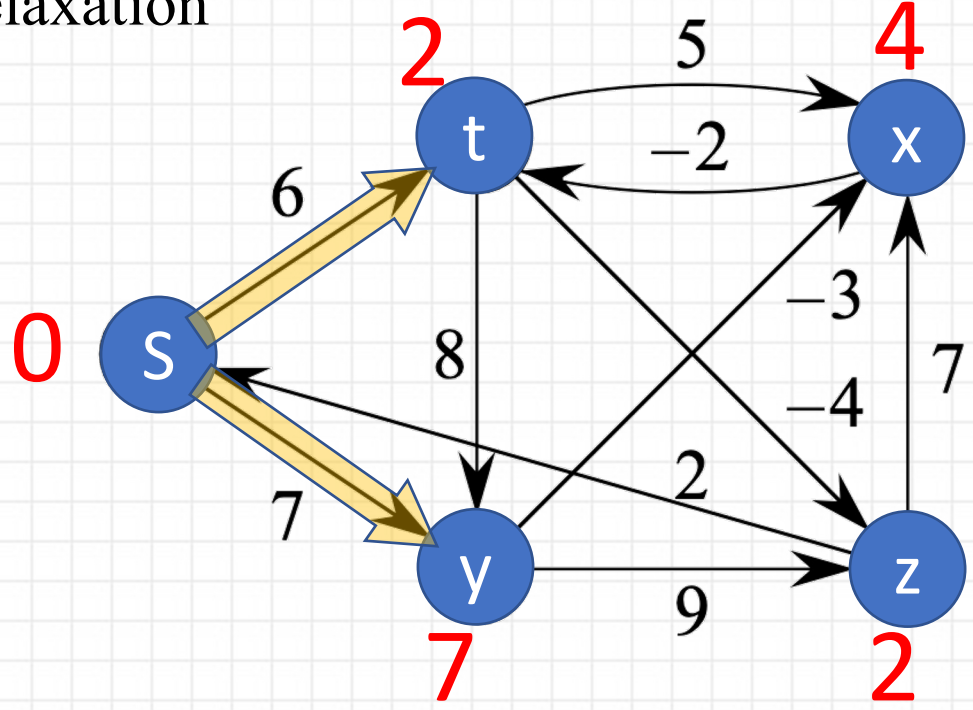
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0				
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=2



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

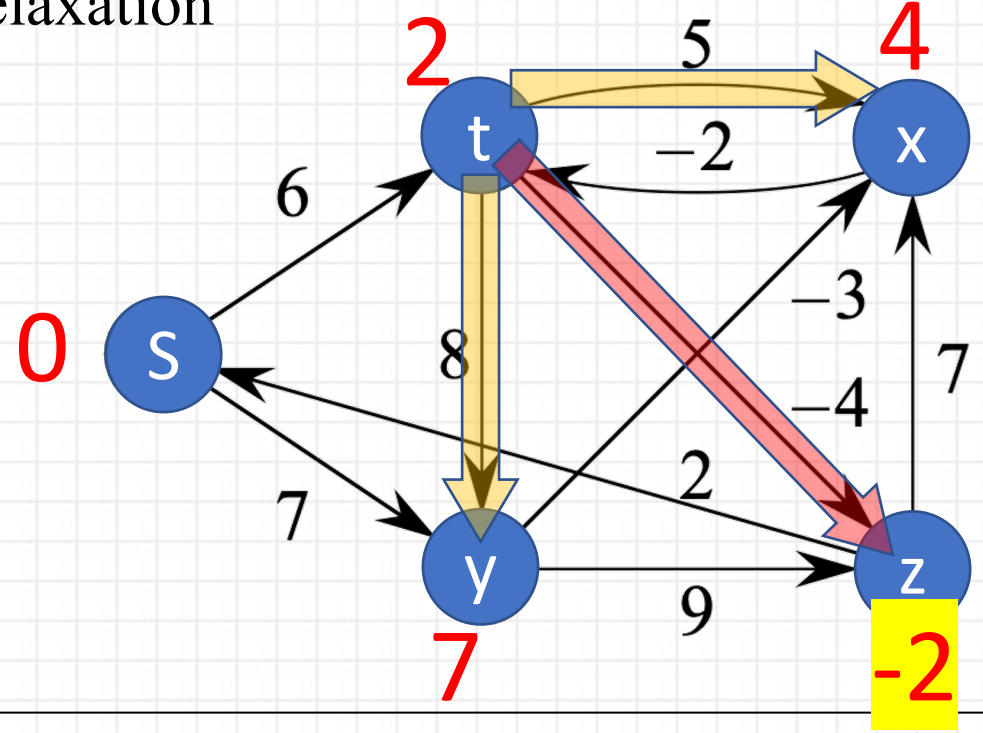
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7		
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=2



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

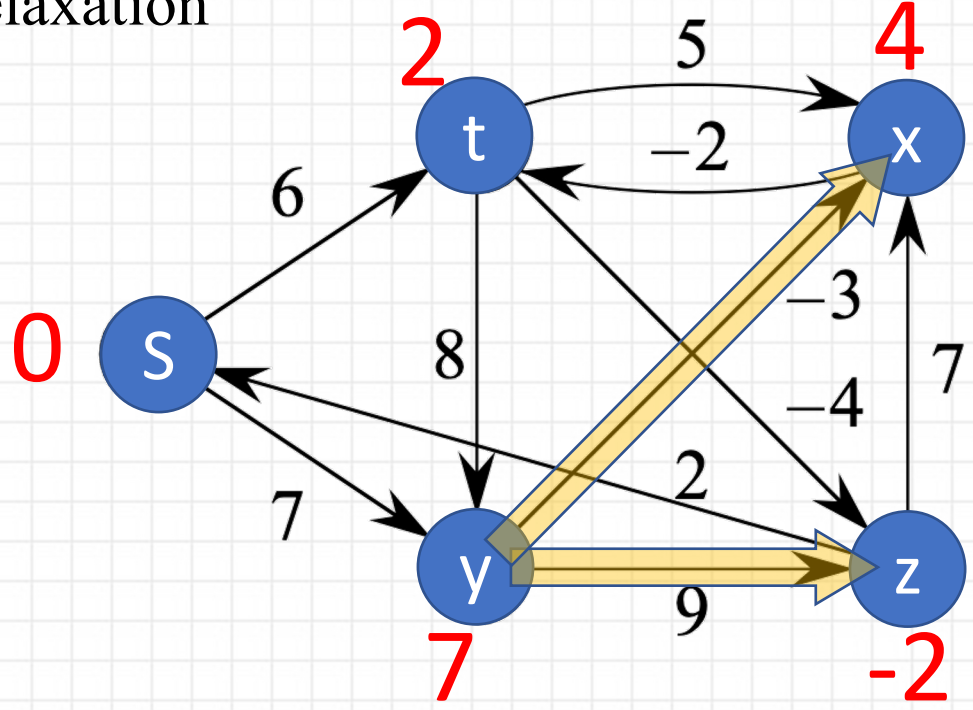
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=2



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

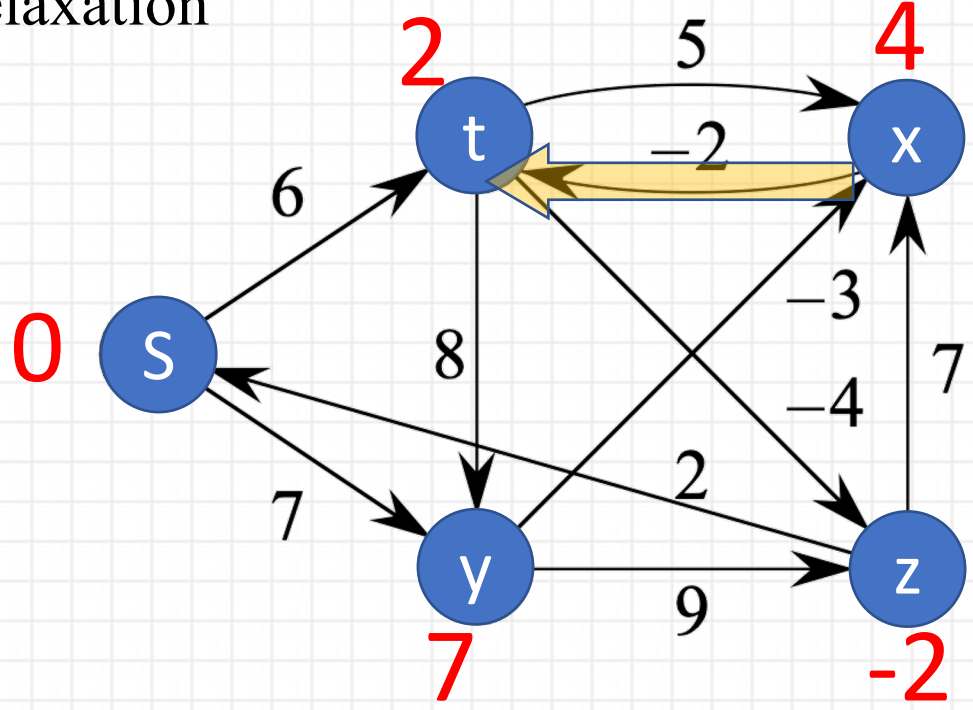
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=2



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

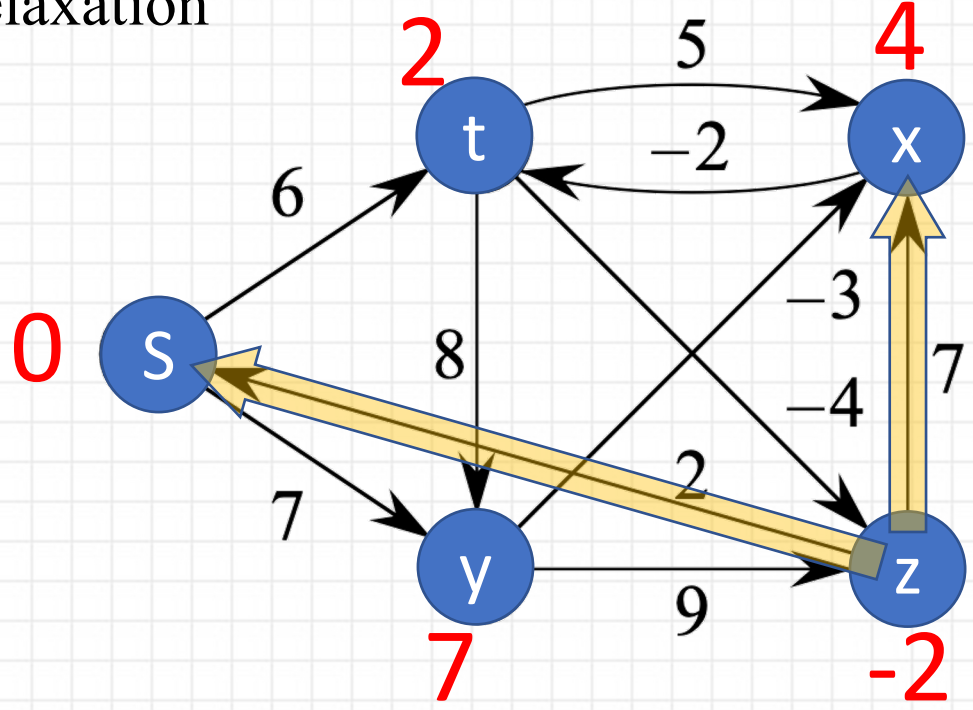
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=2



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

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

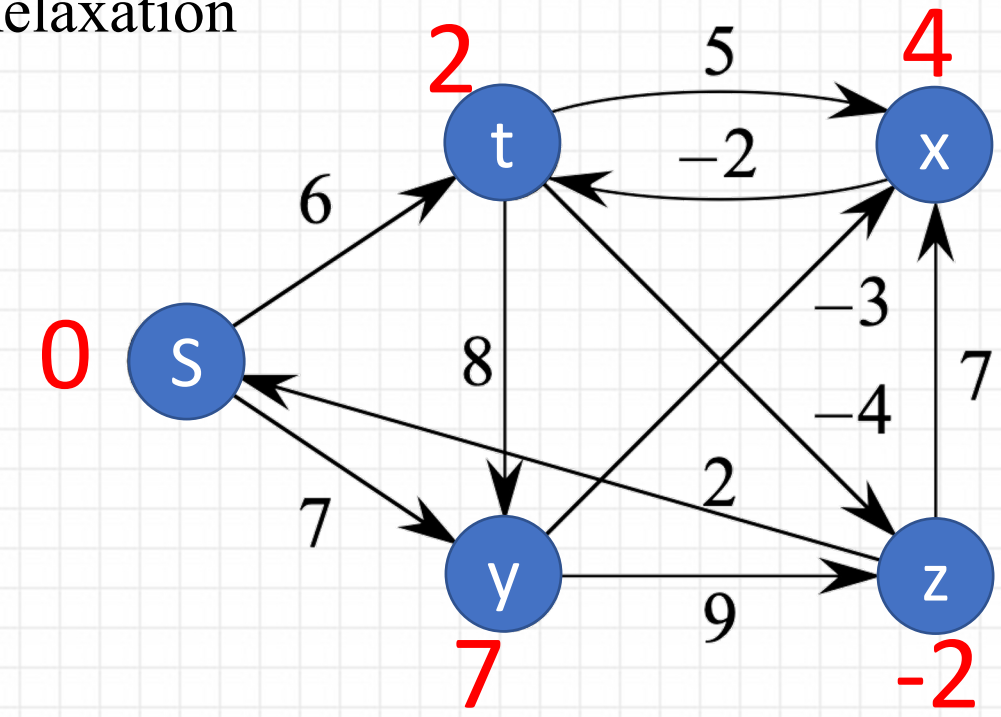
i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3					
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=3

No more updates in iteration 3!



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

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

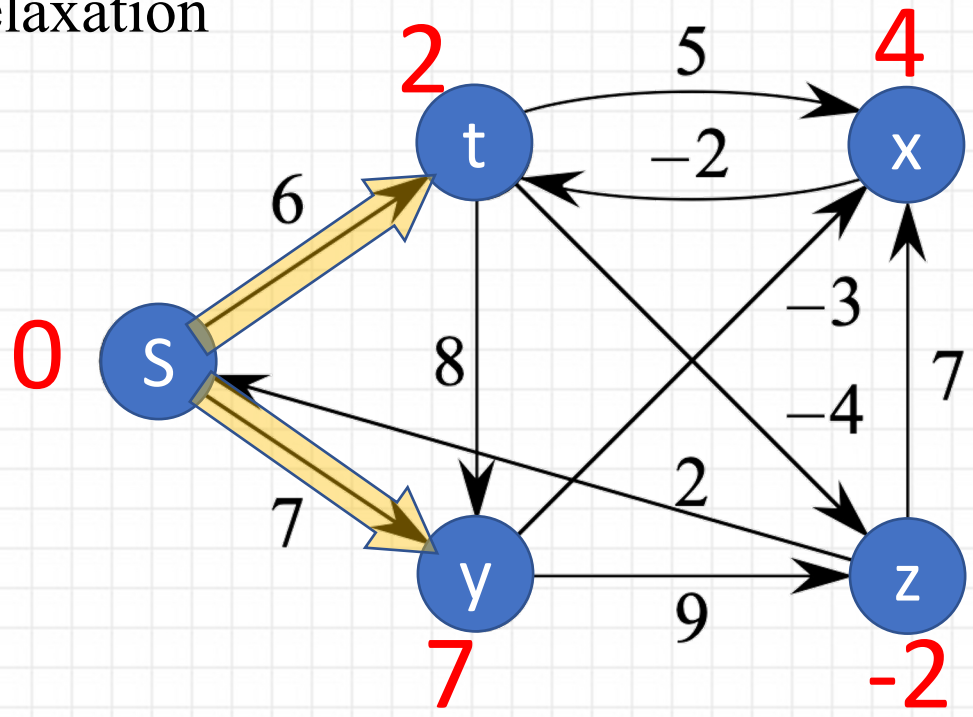
i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0				
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=3

No more updates in iteration 3!



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

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

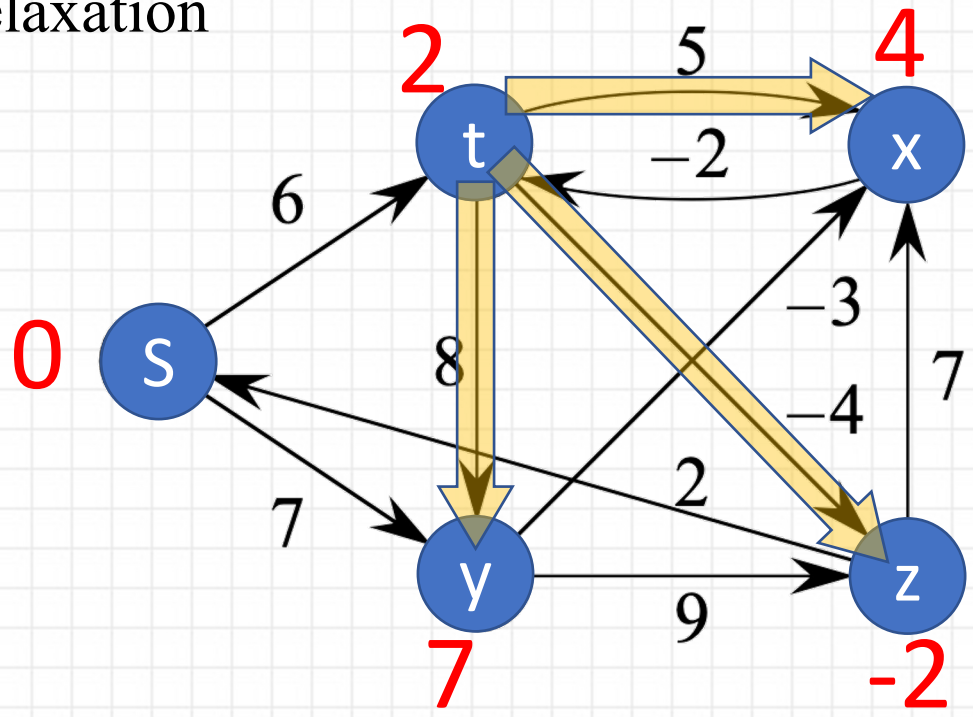
i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0	2	7		
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=3

No more updates in iteration 3!



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

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

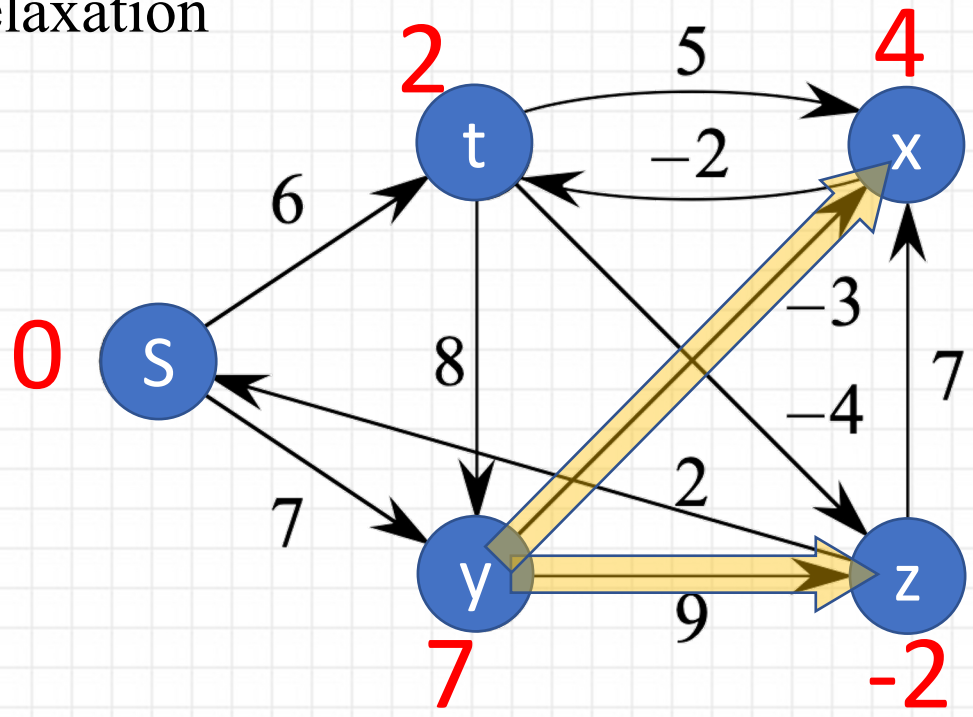
i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0	2	7	4	-2
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=3

No more updates in iteration 3!



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

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

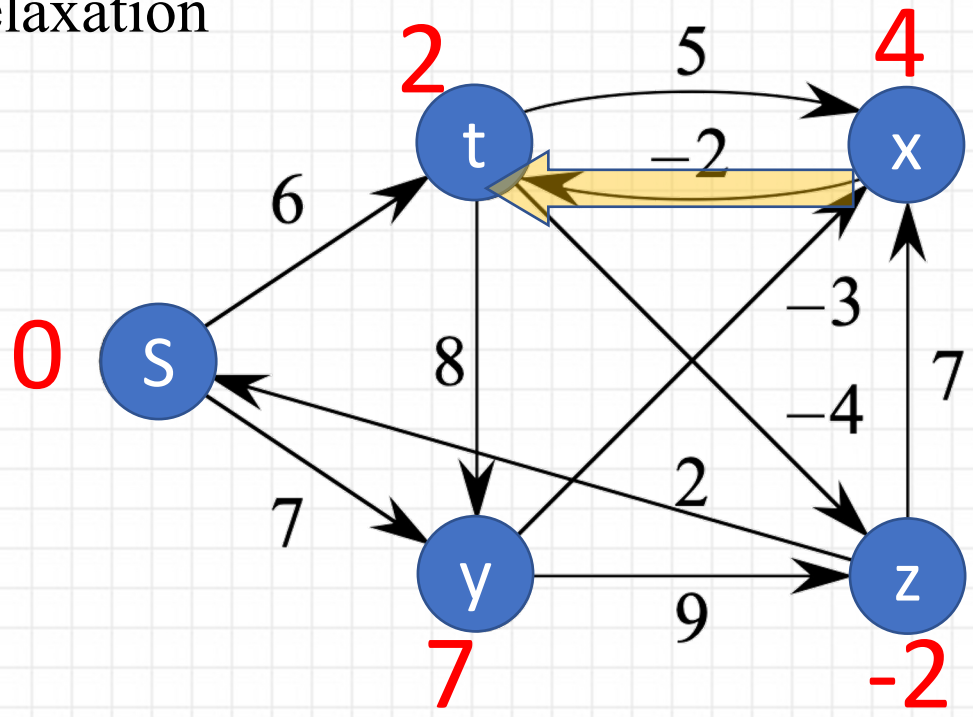
i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0	2	7	4	-2
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=3

No more updates in iteration 3!



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

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

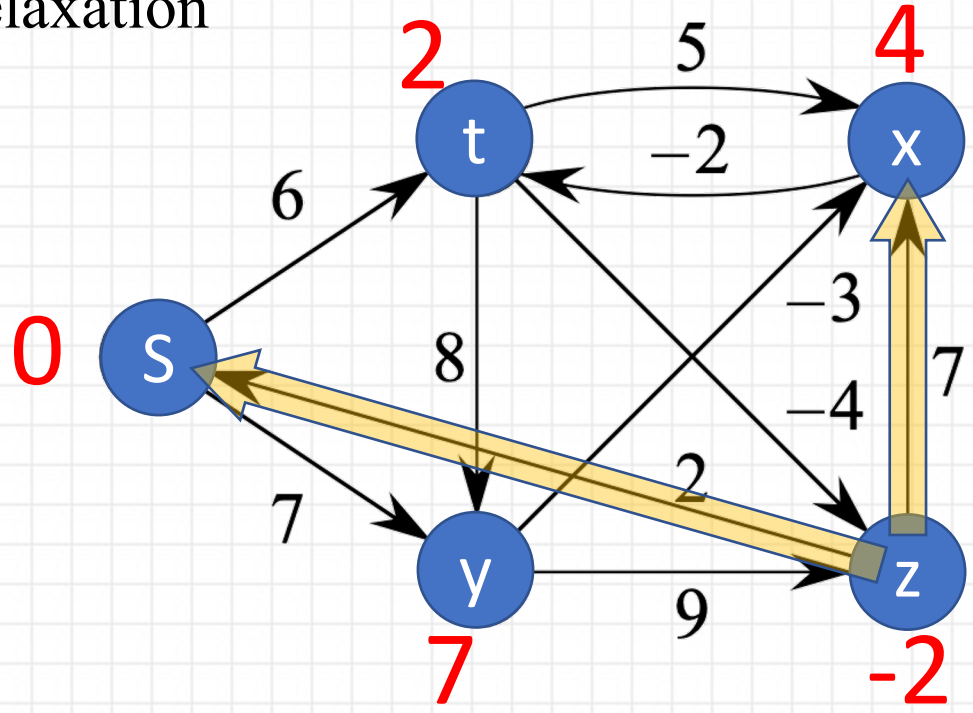
i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0	2	7	4	-2
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=3

No more updates in iteration 3!



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

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

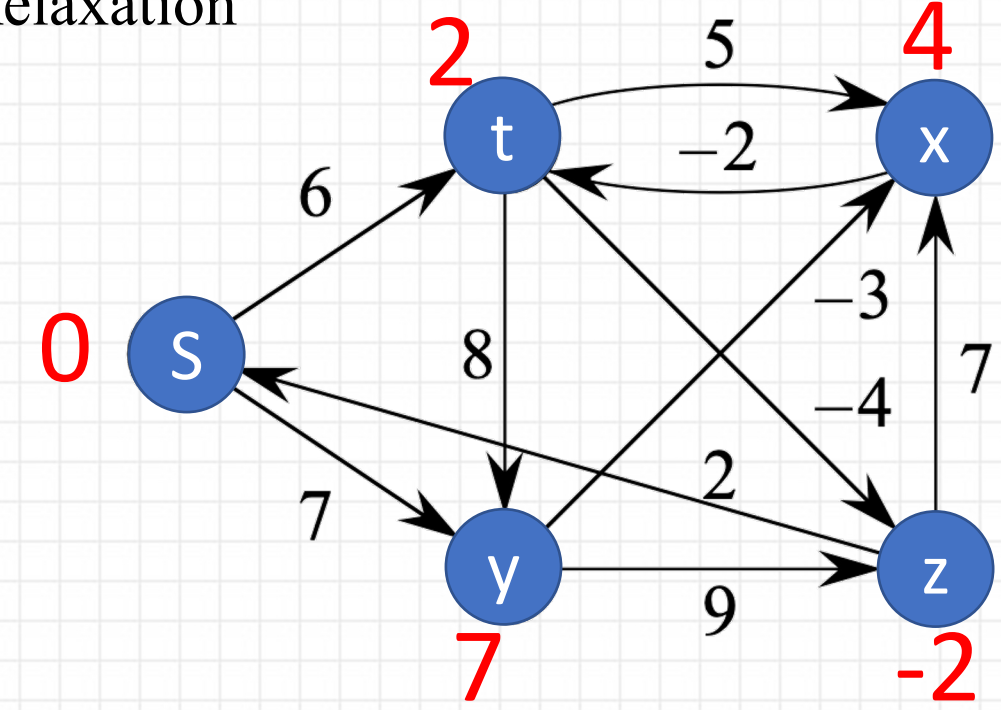
i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0	2	7	4	-2
4					

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation

Iteration i=4

No more updates in iteration 4!



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

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

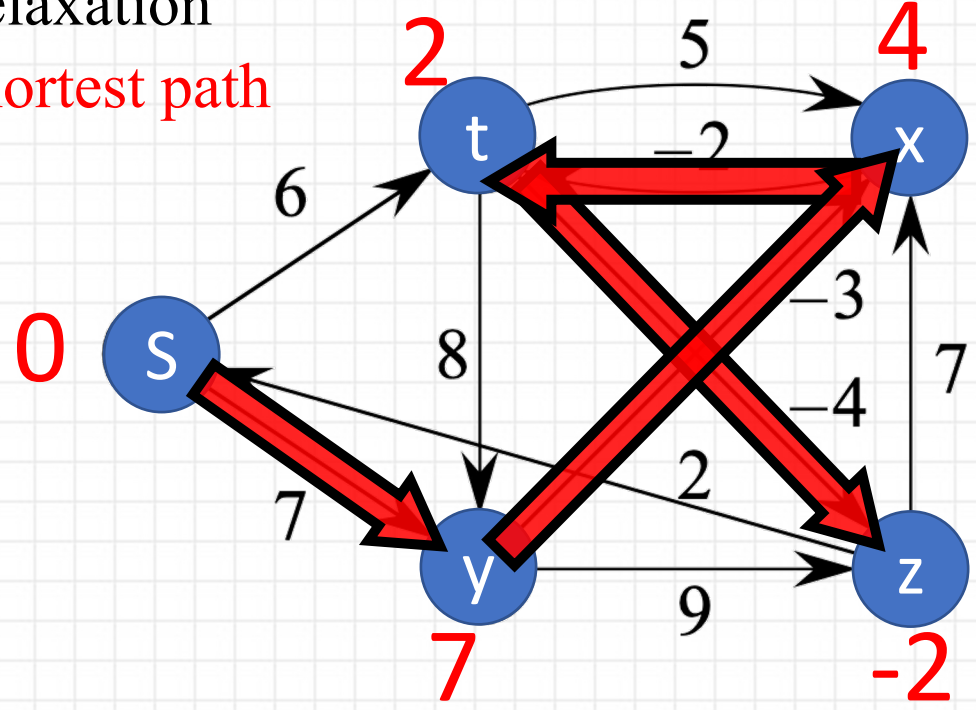
i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0	2	7	4	-2
4	0	2	7	4	-2

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation
  - Shortest path

Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**



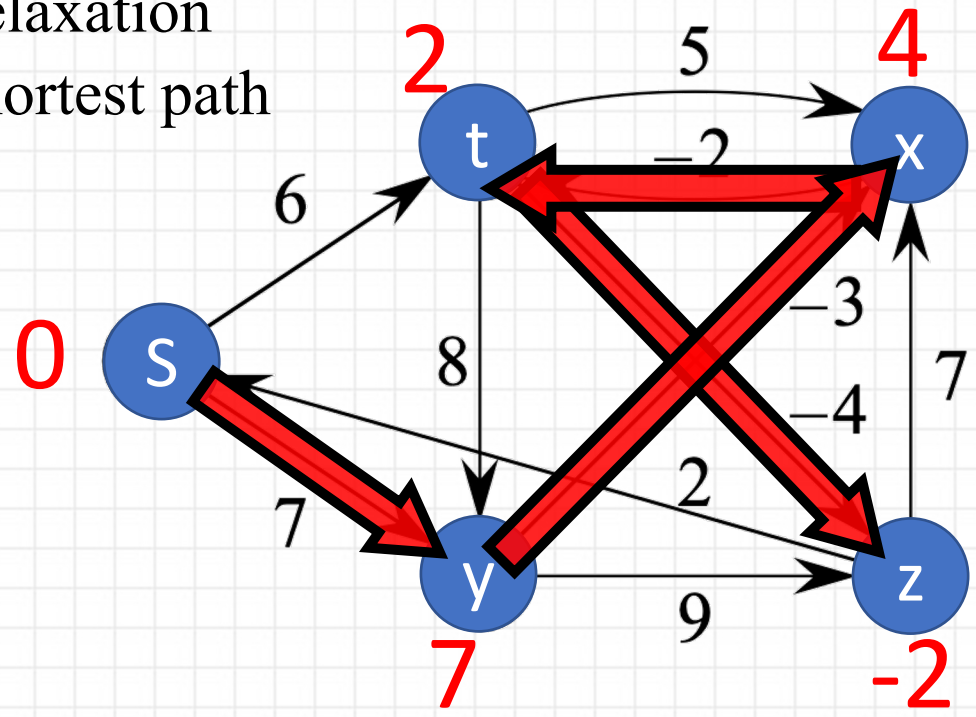

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0	2	7	4	-2
4	0	2	7	4	-2

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation
  - Shortest path

checking for a negative-weight cycle



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

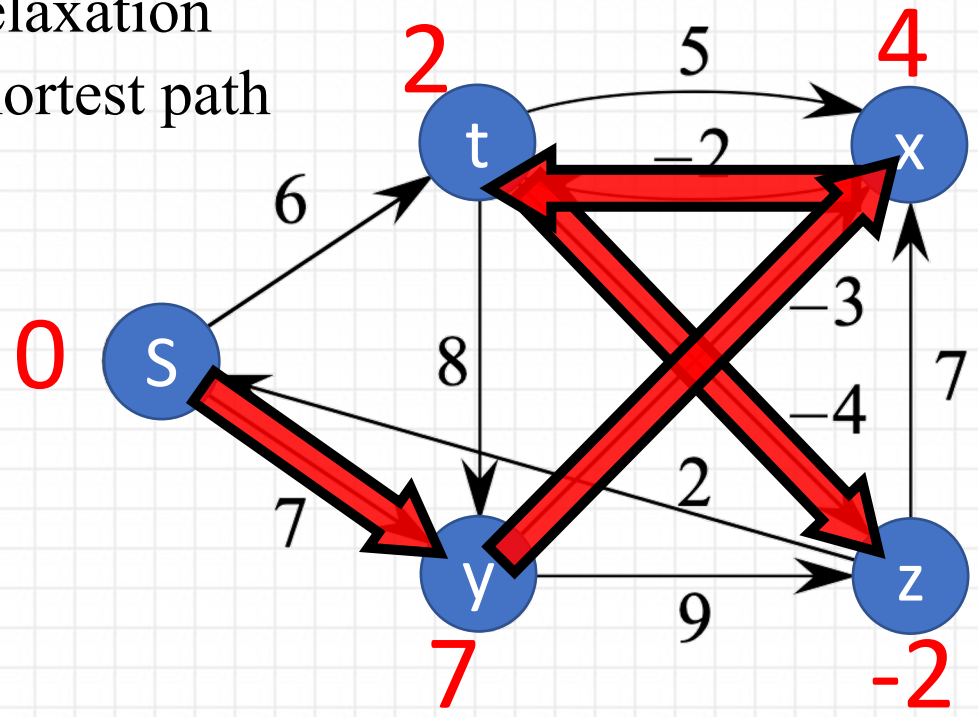
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3        for each edge (u, v) ∈ G.E
4              RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6        if v.d > u.d + w(u, v)
7              return FALSE
8  return TRUE
```

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0	2	7	4	-2
4	0	2	7	4	-2

SSSP: Bellman-Ford

- Example
  - 4 iterations
  - Initialization
  - Relaxation
  - Shortest path

No negative-weight cycle
Return TRUE



Parents (path)

v	π(v)
s	∅
t	**x**
y	**s**
x	**y**
z	**t**

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

i	s	t	y	x	z
0	0	∞	∞	∞	∞
1	0	2	7	4	2
2	0	2	7	4	-2
3	0	2	7	4	-2
4	0	2	7	4	-2

SSSP: Bellman-Ford Summary

- Bellman-Ford
  - Dynamic programming approach
  - Allows negative-weight edges.
  - Computes d[v] and π[v] for all v $\in V$.
  - Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.
  - Running time: $\Theta(|V||E|)$

- How to apply?

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

SSSP: Bellman-Ford Summary

- Bellman-Ford
  - Dynamic programming approach
  - How to apply? (Main steps)

1. Create two tables (both can be implemented using 1D arrays)
   - One for "d", and
   - Another for "π(v)"

Parents (path)

v	π(v)
s	
t	
y	
x	
z	

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

Shortest distance

	s	t	y	x	z
i					

SSSP: Bellman-Ford Summary

- Bellman-Ford
  - Dynamic programming approach
  - How to apply? (Main steps)

1. Create two tables (both can be implemented using 1D arrays)
   - One for "d", and
   - Another for "π(v)"
2. Initialize the tables

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

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

Shortest distance

	s	t	y	x	z
i	0	∞	∞	∞	∞

SSSP: Bellman-Ford Summary

- Bellman-Ford
  - Dynamic programming approach
  - How to apply? (Main steps)

1. Create two tables (both can be implemented using 1D arrays)
   - One for "d", and
   - Another for "π(v)"
2. Initialize the tables
3. Iterate over each node and each time iterate over all edges and update the tables if necessary (relaxation)

$\Theta(|V||E|)$

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

Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

Shortest distance

	s	t	y	x	z
i	0	∞	∞	∞	∞

SSSP: Dijkstra's Algorithm

- Dijkstra
  - Greedy approach
  - No negative-weight edges.
  - Essentially a weighted version of breadth-first search.
  - Instead of a FIFO queue, uses a priority queue.
    Keys are shortest-path weights (d[v]).
  - Have two sets of vertices:
    - S = vertices whose final shortest-path weights are determined,
    - Q = priority queue = $V - S$.

SSSP: Dijkstra's Algorithm

- Dijkstra
  - Greedy approach
  - No negative-weight edges.
  - Essentially a weighted version of BFS
  - Instead of a FIFO queue, uses a priority queue. Keys are shortest-path weights (d[v]).
  - Have two sets of vertices:
    - S = vertices whose final shortest-path weights are determined,
    - Q = priority queue = $V - S$.

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

SSSP: Dijkstra's Algorithm

- Dijkstra
  - Greedy approach
  - No negative-weight *edges*.
  - Essentially a weighted version of BFS
  - Instead of a FIFO queue, uses a priority queue. Keys are shortest-path weights (d[v]).
  - Have two sets of vertices:
    - S = vertices whose final shortest-path weights are determined,
    - Q = priority queue = $V - S$.

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Min priority queue
e.g., binary heap

SSSP: Dijkstra's Algorithm

- Dijkstra's algorithm running time:
  - Like Prim's algorithm, depends on implementation of priority queue.
  - If binary heap, each operation takes $O(\log |V|)$ time $\Rightarrow$ $O(|E| \log |V|)$.
  - If a Fibonacci heap:
    - Each Decrease-Key (used during relaxation) takes $O(1)$ amortized time.
    - There are $O(|V|)$ other operations, taking $O(\log |V|)$ amortized time each.
    - Therefore, time is $O(|V| \log |V| + |E|)$.

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Running time?

SSSP: Dijkstra's Algorithm

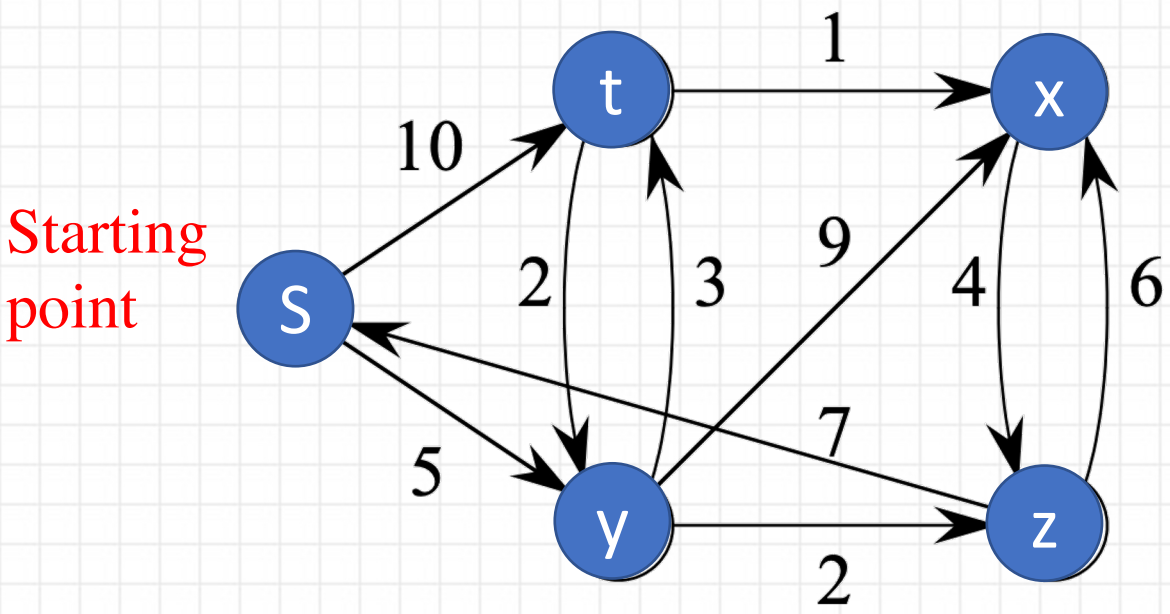
- Dijkstra
  - So, in short,
  - The algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.
  - The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.
  - Greedy strategy:
    Always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S.

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
   then d[v] ← d[u] + w(u, v)
        π[v] ← u
```

SSSP: Dijkstra's Algorithm

- Example



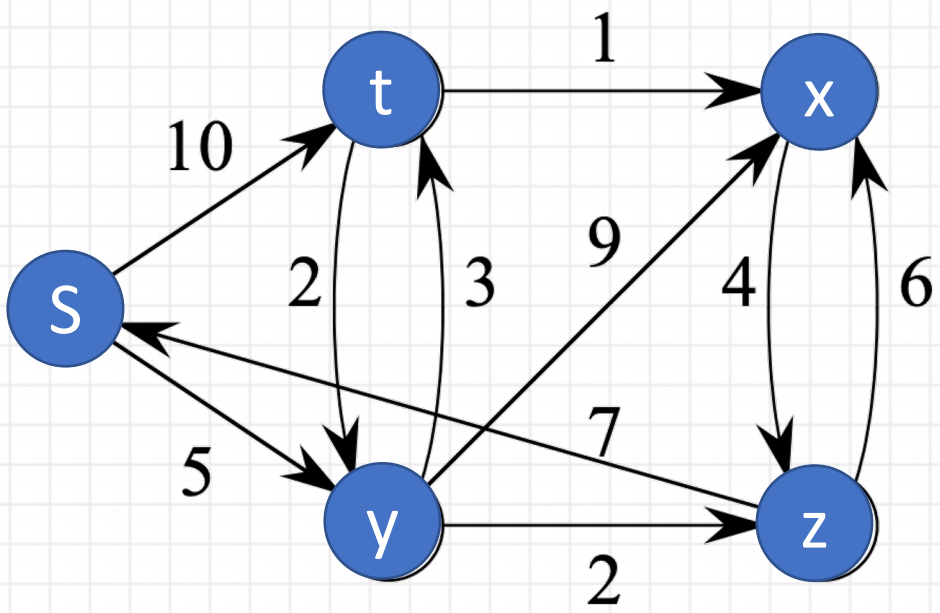

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V            ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

SSSP: Dijkstra's Algorithm

- Example

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```



Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

Shortest distance

s	t	y	x	z
0	∞	∞	∞	∞

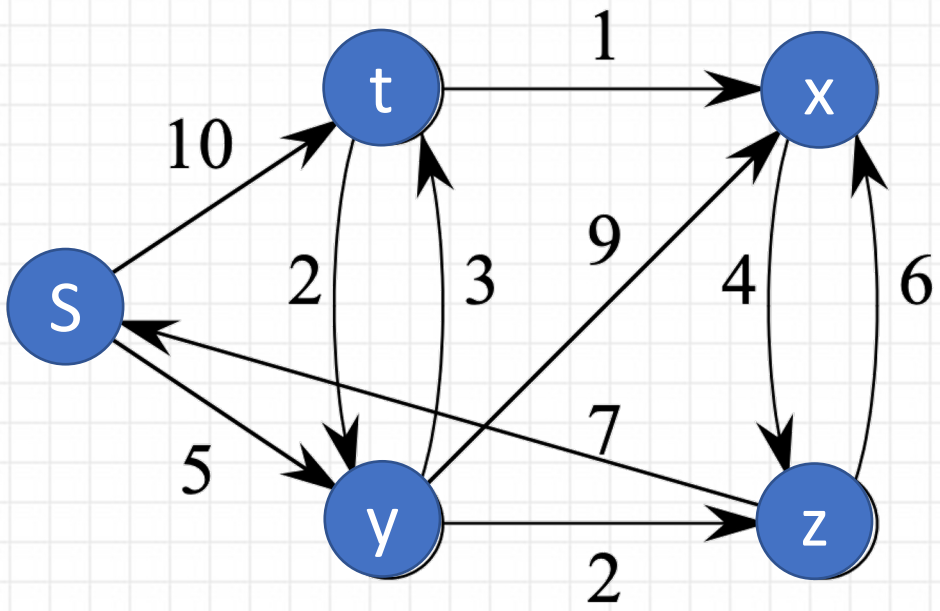
SSSP: Dijkstra's Algorithm

- Example
  - S = {}
  - Q = {s: 0, t: ∞, y: ∞, x: ∞, z: ∞}

unvisited nodes (priority queue)

Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅




```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Shortest distance

s	t	y	x	z
0	∞	∞	∞	∞

SSSP: Dijkstra's Algorithm

- Example
  - S = {}
  - Q = {s: 0, t: ∞, y: ∞, x: ∞, z: ∞}

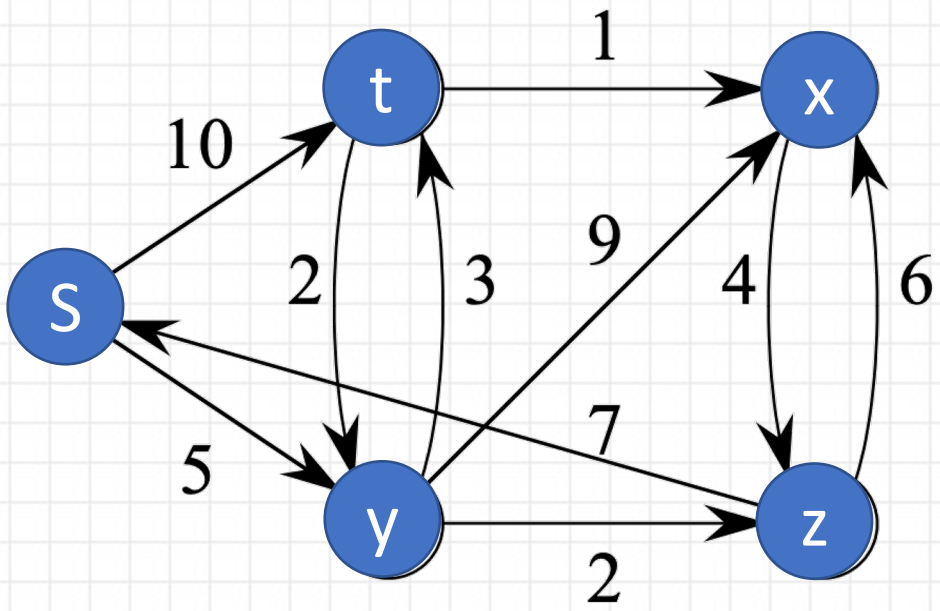
U←Extract-Min(Q)

unvisited nodes (priority queue)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅



Shortest distance

s	t	y	x	z
0	∞	∞	∞	∞

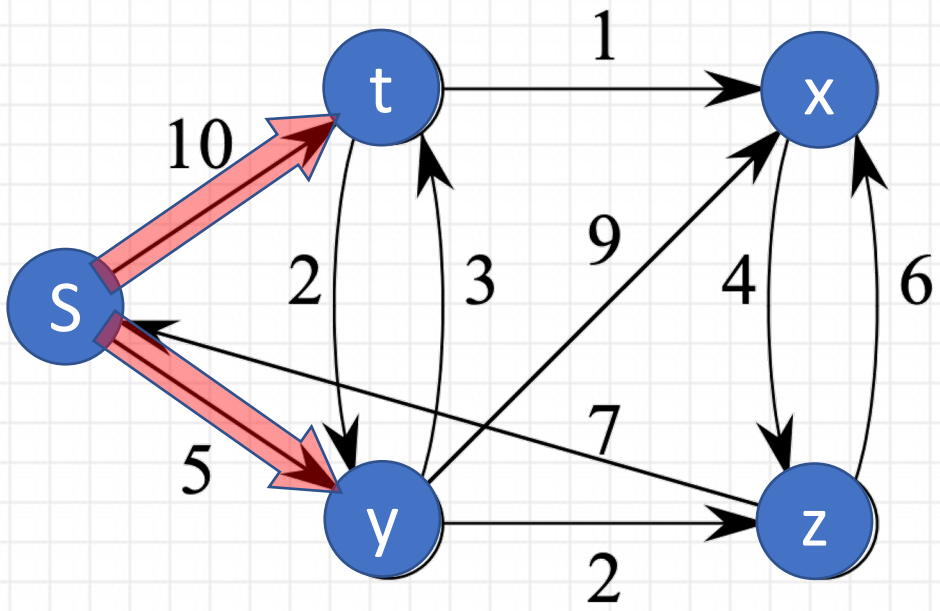
SSSP: Dijkstra's Algorithm

- Example
  - S = {s}
  - Q = {~~s: 0~~, t: 10, y: 5, x: ∞, z: ∞}

unvisited nodes (priority queue)

Parents (path)

v	π(v)
s	∅
t	s
y	s
x	∅
z	∅




```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Shortest distance

s	t	y	x	z
0	10	5	∞	∞

SSSP: Dijkstra's Algorithm

- Example
  - S = {s}
  - Q = {~~s: 0~~, t: 10, y: 5, x: ∞, z: ∞}

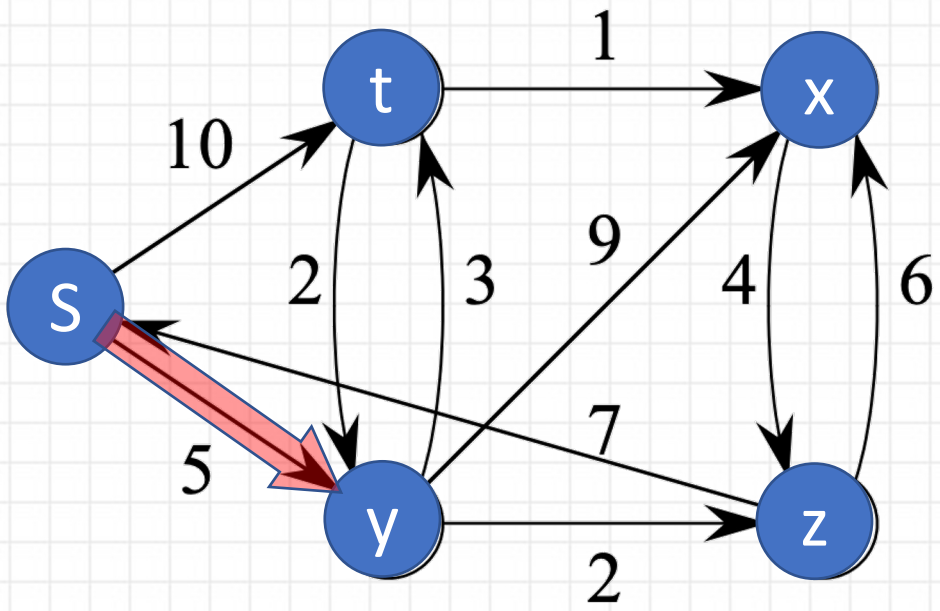
unvisited nodes (priority queue)

U←Extract-Min(Q)
(The smallest that has not been chosen yet)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Parents (path)

v	π(v)
s	∅
t	s
y	s
x	∅
z	∅



Shortest distance

s	t	y	x	z
0	10	5	∞	∞

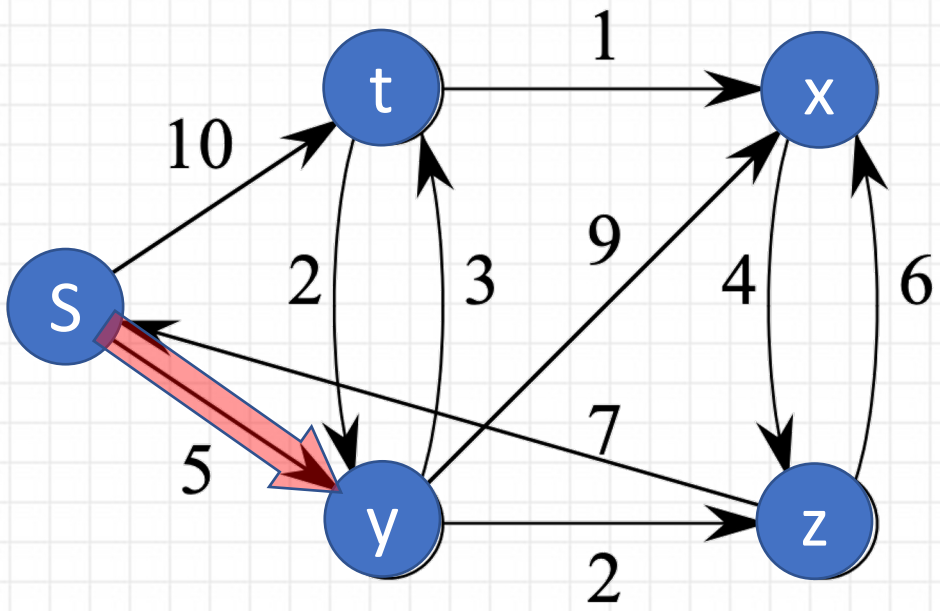
SSSP: Dijkstra's Algorithm

- Example
  - S = {s, y}
  - Q = {~~s: 0~~, t: 10, ~~y: 5~~, x: ∞, z: ∞}

unvisited nodes (priority queue)

Parents (path)

v	π(v)
s	∅
t	s
y	s
x	∅
z	∅




```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Shortest distance

s	t	y	x	z
0	10	5	∞	∞

SSSP: Dijkstra's Algorithm

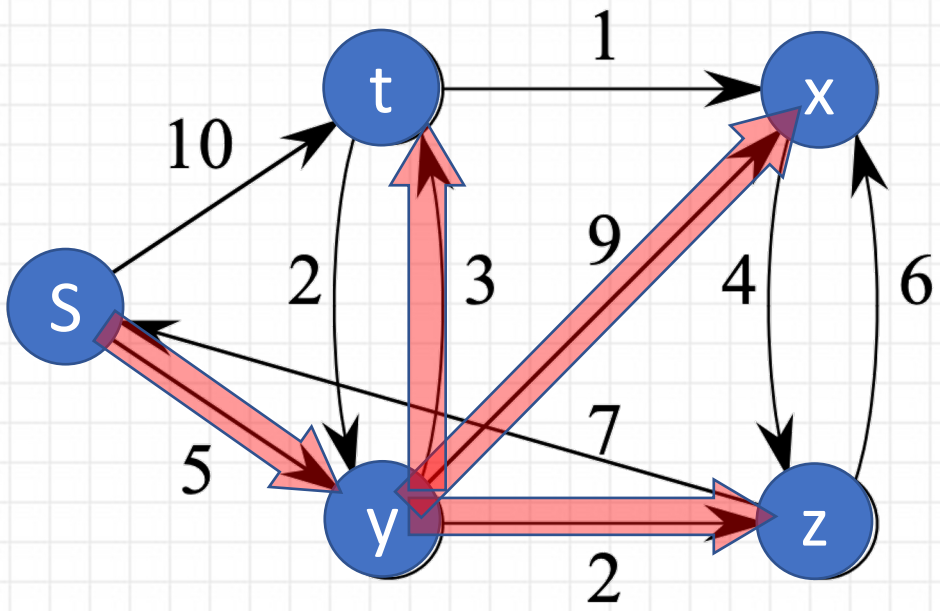
- Example
  - S = {s, y}
  - Q = {~~s: 0~~, t: 8, ~~y: 5~~, x: 14, z: 7}

unvisited nodes (priority queue)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	y
z	y



Shortest distance

s	t	y	x	z
0	8	5	14	7

SSSP: Dijkstra's Algorithm

- Example
  - S = {s, y}
  - Q = {~~s: 0~~, t: 8, ~~y: 5~~, x: 14, z: 7}

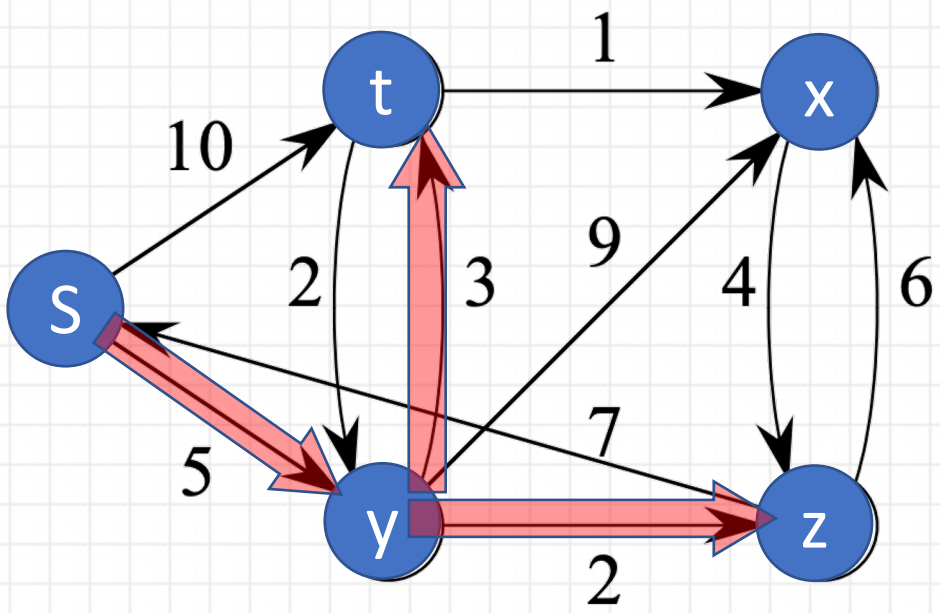
U←Extract-Min(Q)
(The smallest that has not been chosen yet)

unvisited nodes (priority queue)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	y
z	y



Shortest distance

s	t	y	x	z
0	8	5	14	7

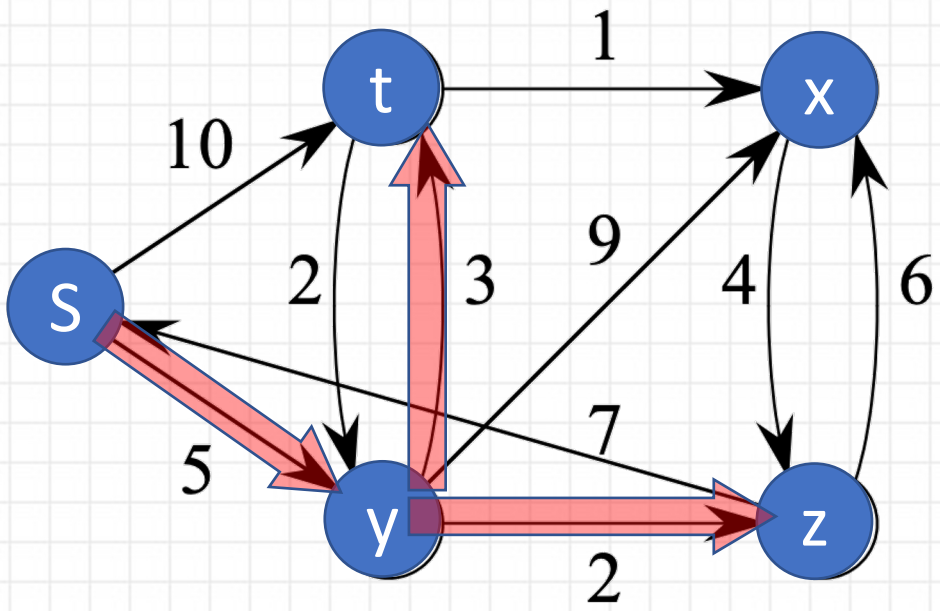
SSSP: Dijkstra's Algorithm

- Example
  - S = {s, y, z}
  - Q = {~~s: 0~~, t: 8, ~~y: 5~~, x: 14, ~~z: 7~~}

unvisited nodes (priority queue)

Parents (path)

v	π(v)
s	∅
t	**y**
y	**s**
x	**y**
z	**y**




```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Shortest distance

s	t	y	x	z
0	8	5	14	7

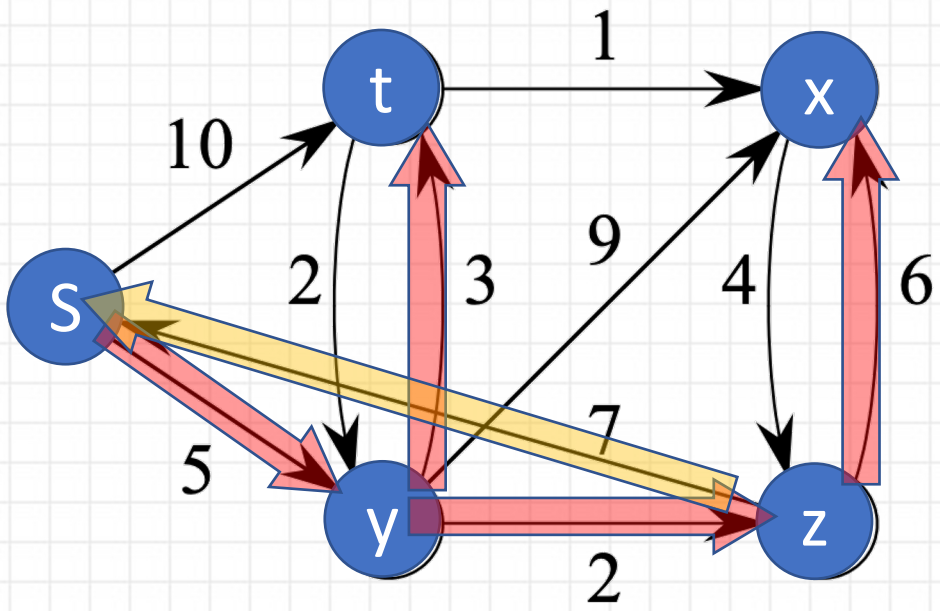
SSSP: Dijkstra's Algorithm

- Example
  - S = {s, y, z}
  - Q = {~~s: 0~~, t: 8, ~~y: 5~~, x: 13, ~~z: 7~~}

unvisited nodes (priority queue)

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	z
z	y




```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Shortest distance

s	t	y	x	z
0	8	5	13	7

SSSP: Dijkstra's Algorithm

- Example
  - S = {s, y, z}
  - Q = {~~s: 0~~, t: 8, ~~y: 5~~, x: 13, ~~z: 7~~}

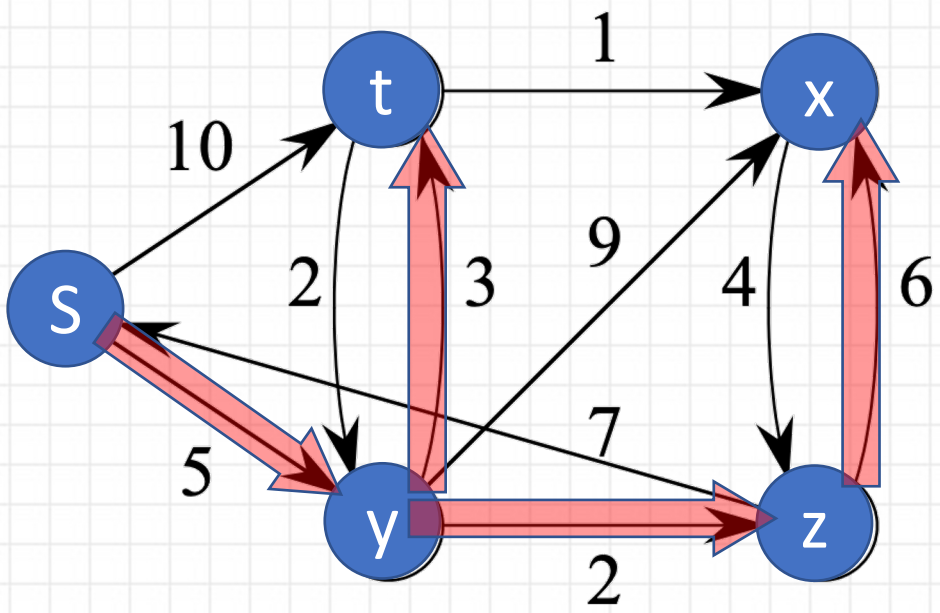
unvisited nodes (priority queue)

U←Extract-Min(Q)
(The smallest that has not been chosen yet)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	z
z	y



Shortest distance

s	t	y	x	z
0	8	5	13	7

SSSP: Dijkstra's Algorithm

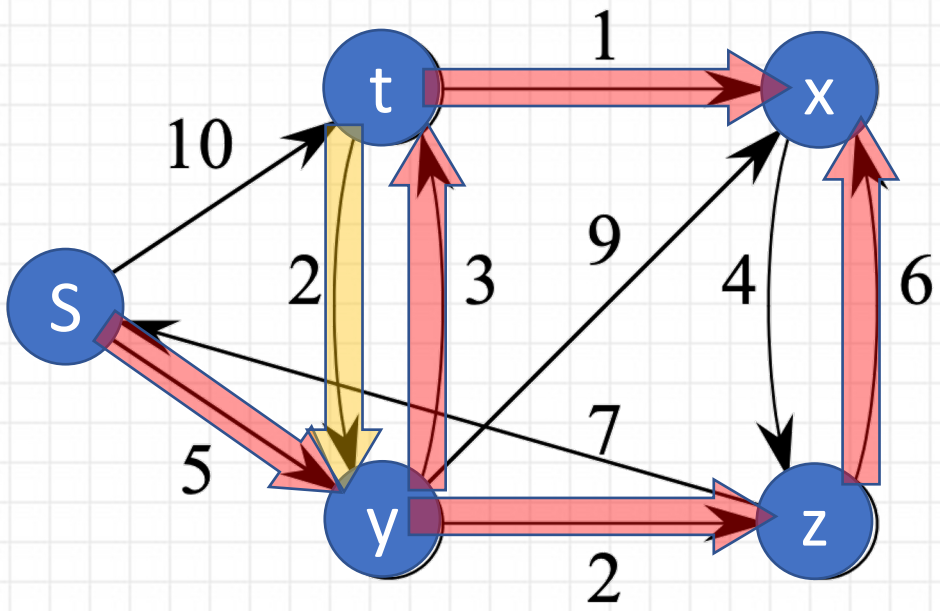
- Example
  - S = {s, y, z, t}
  - Q = {~~s: 0~~, ~~t: 8~~, ~~y: 5~~, x: 9, ~~z: 7~~}

unvisited nodes (priority queue)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	t
z	y



Shortest distance

s	t	y	x	z
0	8	5	9	7

SSSP: Dijkstra's Algorithm

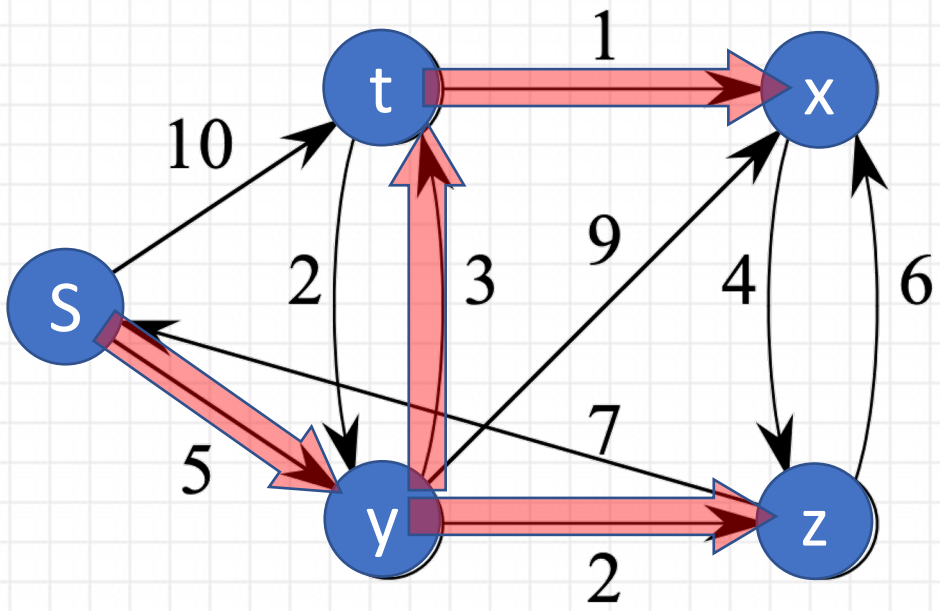
- Example
  - S = {s, y, z, t}
  - Q = {~~s: 0~~, ~~t: 8~~, ~~y: 5~~, x: 9, ~~z: 7~~}

unvisited nodes (priority queue)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	t
z	y



Shortest distance

s	t	y	x	z
0	8	5	9	7

SSSP: Dijkstra's Algorithm

- Example
  - S = {s, y, z, t}
  - Q = {~~s: 0~~, ~~t: 8~~, ~~y: 5~~, x: 9, ~~z: 7~~}

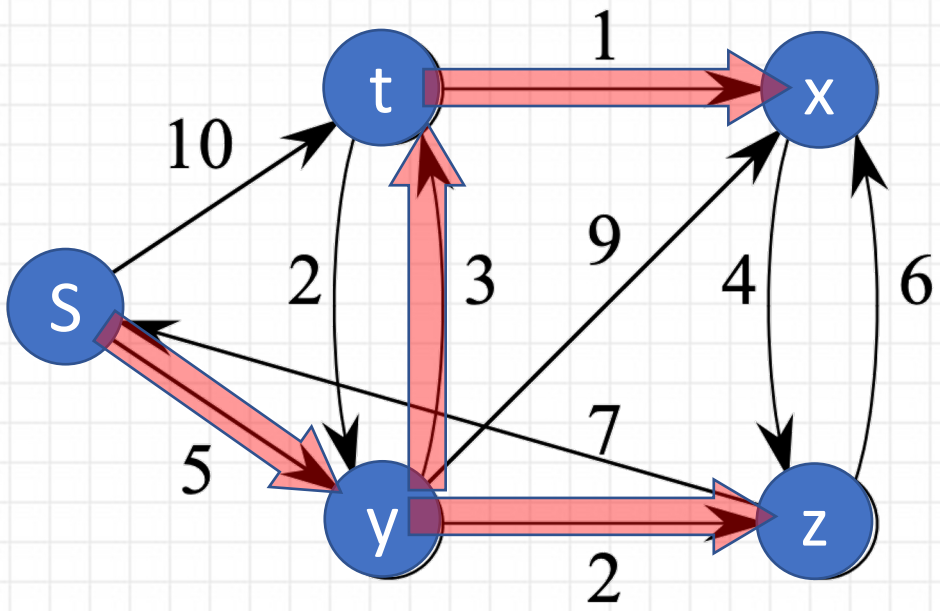
unvisited nodes (priority queue)

U←Extract-Min(Q)
(The smallest that has not been chosen yet)

DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
$S \leftarrow \emptyset$
$Q \leftarrow V$ ▷ i.e., insert all vertices into Q
while $Q \neq \emptyset$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each vertex $v \in Adj[u]$
      do RELAX(u, v, w)

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	t
z	y



Shortest distance

s	t	y	x	z
0	8	5	9	7

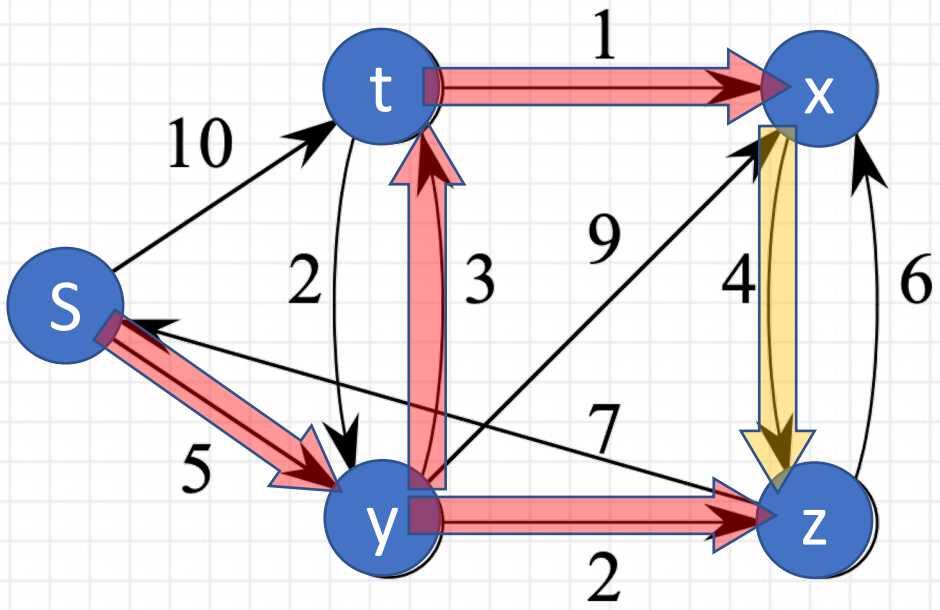
SSSP: Dijkstra's Algorithm

- Example
  - S = {s, y, z, t, x}
  - Q = {~~s: 0~~, ~~t: 8~~, ~~y: 5~~, ~~x: 9~~, ~~z: 7~~}

unvisited nodes (priority queue)

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	t
z	y




```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Shortest distance

s	t	y	x	z
0	8	5	9	7

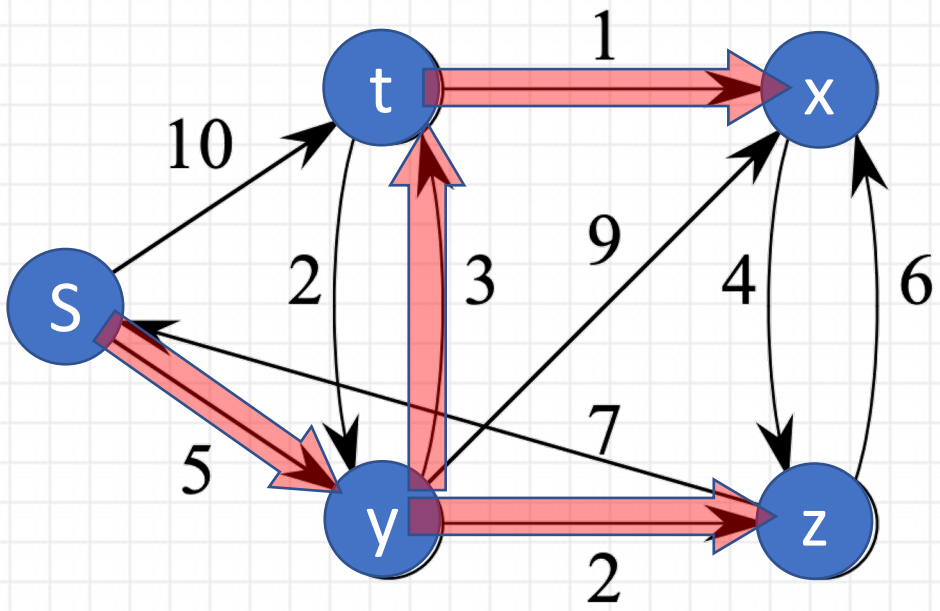
SSSP: Dijkstra's Algorithm

- Example
  - S = {s, y, z, t, x}
  - Q = {~~s: 0~~, ~~t: 8~~, ~~y: 5~~, ~~x: 9~~, ~~z: 7~~}

unvisited nodes (priority queue)

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	t
z	y




```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Shortest distance

s	t	y	x	z
0	8	5	9	7

SSSP: Dijkstra's Algorithm

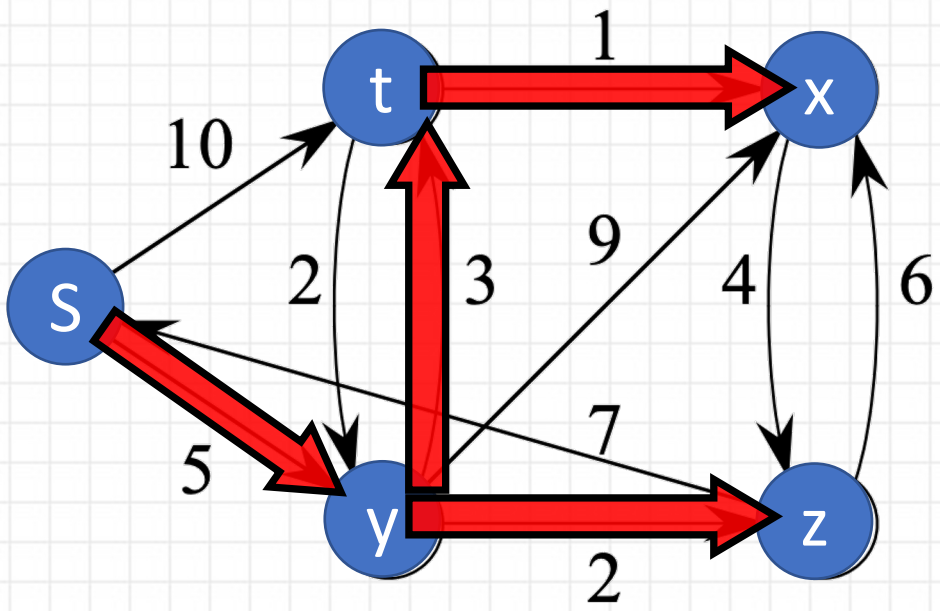
- Example
  - S = {s, y, z, t, x}
  - Q = {~~s: 0~~, ~~t: 8~~, ~~y: 5~~, ~~x: 9~~, ~~z: 7~~}

unvisited nodes (priority queue)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Parents (path)

v	π(v)
s	∅
t	y
y	s
x	t
z	y



Shortest distance

s	t	y	x	z
0	8	5	9	7

SSSP: Dijkstra's Algorithm Summary

- Dijkstra
  - Greedy approach
  - No negative-weight edges.
  - Essentially a weighted version of BFS
  - Instead of a FIFO queue, uses a priority queue. Keys are shortest-path weights (d[v]).
  - If binary heap, each operation takes $O(\log |V|)$ time $\Rightarrow O(|E| \log |V|)$.

  - How to apply? (Main steps)

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

SSSP: Dijkstra's Algorithm Summary

- Dijkstra
  - Greedy approach
  - No negative-weight edges.
  - How to apply? (Main steps)

1. Create three data structure
   - One for the priority queue Q (usually min binary heap)
   - One for "d" shortest path weight estimation
   - Another for "π(v)"
2. Initialize them

```
INIT-SINGLE-SOURCE(V, s)
for each v ∈ V
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Q = {s: 0, t: ∞, y: ∞, x: ∞, z: ∞}

Shortest distance

s	t	y	x	z
0	∞	∞	∞	∞

Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

SSSP: Dijkstra's Algorithm Summary

- Dijkstra
  - Greedy approach
  - No negative-weight edges.
  - How to apply? (Main steps)

1. Create three data structure
   - One for the priority queue Q
   - One for "d" shortest path weight estimation
   - Another for "π(v)"
2. Initialize them
3. At each step deque the min-distance not chosen node u from the priority queue, and update the neighbors (relax) and the key of the priority Q.

$O(|E| \log |V|)$.

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Q = {s: 0, t: ∞, y: ∞, x: ∞, z: ∞}

Shortest distance

s	t	y	x	z
0	∞	∞	∞	∞

Parents (path)

v	π(v)
s	∅
t	∅
y	∅
x	∅
z	∅

All-Pairs Shortest Path (APSP)

- Problem description
  - Given graph $G = (V, E)$, and a weight function w: $E \rightarrow \mathbb{R}$
  - Output: An $n \times n$ matrix of shortest path distances $\delta(u, v)$.

- Can we use Bellman-Ford or Dijkstra's algorithms?
  - Running Bellman-Ford once from each vertex:
    - $O(|V|^2|E|)$ which is $O(|V|^4)$ if the graph is *dense* ($|E| = \Theta(|V|^2)$).
  - If non-negative weights, then we can run Dijkstra's algorithm once from each vertex:
    - $O(|V||E| \log |V|)$ with binary heap – $O(|V|^3)$ if dense,
    - $O(|V|^2 \log|V| + |V||E|)$ with Fibonacci heap – $O(|V|^3)$ if dense.

All-Pairs Shortest Path (APSP)

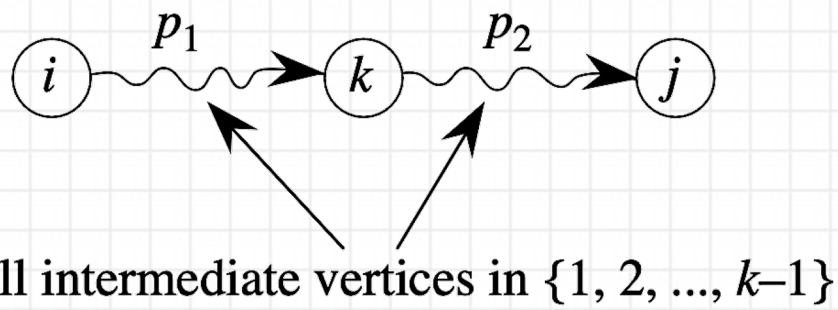
- Problem description
  - Given graph $G = (V, E)$, and a weight function $w: E \to \mathbb{R}$
  - Output: An $n \times n$ matrix of shortest path distances $\delta(u, v)$.

Let's see how we can do it in $O(|V|^3)$ without any fancy data structure.

- Can we use Bellman-Ford or Dijkstra's algorithms?
  - Running Bellman-Ford once from each vertex:
    - $O(|V|^2|E|)$ which is $O(|V|^4)$ if the graph is *dense* ($|E| = \Theta(|V|^2)$).
  - If non-negative weights, then we can run Dijkstra's algorithm once from each vertex:
    - $O(|V||E| \log |V|)$ with binary heap – $O(|V|^3)$ if dense,
    - $O(|V|^2 \log|V| + |V||E|)$ with Fibonacci heap – $O(|V|^3)$ if dense.

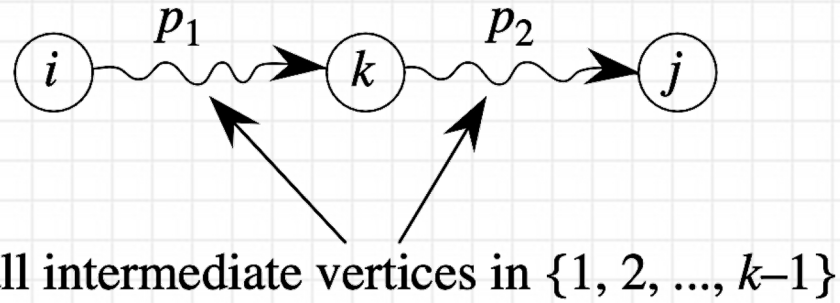
APSP: Floyd-Warshall

- Floyd-Warshall algorithm
  - Let $d_{ij}^{(k)}$ be the shortest path weight of any path from i to j with all intermediate vertices in $\{1, \ldots, k\}$.
  - Ultimately, we need to find the values of $d_{ij}^{(n)}$ for each pair of nodes v_i and v_j.
  - Consider a shortest path p (i $\rightsquigarrow$ j), from i to j, with all intermediate vertices in $\{1, 2, \ldots, k\}$, then 2 cases can happen:
    1. If k is not an intermediate vertex, then all intermediate vertices of p are in $\{1, \ldots, k-1\}$.
    2. If k is an intermediate vertex,
       then p is composed of two shortest sub-paths,
       with the intermediate nodes drawn from $\{1, \ldots, k-1\}$



APSP: Floyd-Warshall

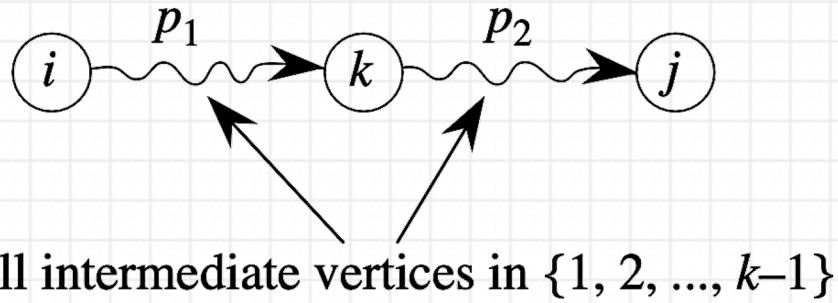
- Floyd-Warshall algorithm
  - Consider a shortest path p (i ⇝ j), from i to j, with all intermediate vertices in $\{1, 2, \ldots, k\}$, then 2 cases can happen:
    1. If k is not an intermediate vertex, then all intermediate vertices of p are in $\{1, \ldots, k-1\}$.
    2. If k is an intermediate vertex,
       then p is composed of two shortest sub-paths,
       with the intermediate nodes drawn from $\{1, \ldots, k-1\}$
       (optimal substructure of shortest path)



  - Base case $d_{ij}^{(0)} = W_{ij}$
    (no intermediate vertices)

APSP: Floyd-Warshall

- Floyd-Warshall algorithm
  - Consider a shortest path p (i ⇝ j), from i to j, with all intermediate vertices in $\{1, 2, \ldots, k\}$, then 2 cases can happen:
    1. If k is not an intermediate vertex, then all intermediate vertices of p are in $\{1, \ldots, k-1\}$.
    2. If k is an intermediate vertex,
       then p is composed of two shortest sub-paths,
       with the intermediate nodes drawn from $\{1, \ldots, k-1\}$
       (optimal substructure of shortest path)



  - Base case $d_{ij}^{(0)} = W_{ij}$
    (no intermediate vertices)

We will use a weight matrix W which is defined as:

$$W_{ij} = \begin{cases} 0 & i = j \\ w(i,j) & i \neq j \text{ and } (i,j) \in E \\ \infty & i \neq j \text{ and } (i,j) \notin E \end{cases}$$

APSP: Floyd-Warshall

- Floyd-Warshall algorithm
  - Dynamic programming approach
  - We will use a weight matrix W which is defined as: $$W_{ij} = \begin{cases} 0 & i = j \\ w(i,j) & i \neq j \; and \; (i,j) \in E \\ \infty & i \neq j \; and \; (i,j) \notin E \end{cases}$$
  - Recurrence relation: $$d_{ij}^{(k)} = \begin{cases} W_{ij} & k = 0 \\ \min \begin{cases} d_{ij}^{(k-1)} \\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} & k \geq 1 \end{cases}$$

APSP: Floyd-Warshall

- Floyd-Warshall algorithm
  - Dynamic programming approach
  - We will use a weight matrix W which is defined as:

$$W_{ij} = \begin{cases} 0 & i = j \\ w(i,j) & i \neq j \; and \; (i,j) \in E \\ \infty & i \neq j \; and \; (i,j) \notin E \end{cases}$$

  - Recurrence relation:

$$d_{ij}^{(k)} = \begin{cases} W_{ij} & k = 0 \\ \min \begin{cases} d_{ij}^{(k-1)} \\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} & k \geq 1 \end{cases}$$

We want $D^{(n)} = d_{ij}^{(n)}$

APSP: Floyd-Warshall

- Floyd-Warshall algorithm

We want $D^{(n)} = d_{ij}^{(n)}$

  - Recurrence relation:

$$d_{ij}^{(k)} = \begin{cases} W_{ij} & k = 0 \\ \min \begin{cases} d_{ij}^{(k-1)} \\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} & k \geq 1 \end{cases}$$

  - Implementation:
    - Bottom-up (iterative)

```
FLOYD-WARSHALL(W, n)
D^(0) ← W
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

APSP: Floyd-Warshall

- Floyd-Warshall algorithm

We want $D^{(n)} = d_{ij}^{(n)}$

  - Recurrence relation:

$$d_{ij}^{(k)} = \begin{cases} W_{ij} & k = 0 \\ \min \begin{cases} d_{ij}^{(k-1)} \\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} & k \geq 1 \end{cases}$$

  - Implementation:
    - Bottom-up (iterative)

```
FLOYD-WARSHALL(W, n)
D^(0) ← W
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

  - Running time?

APSP: Floyd-Warshall

- Floyd-Warshall algorithm
  - Implementation:
    - Bottom-up (iterative)
  - Running time?
    - $O(|V|^3)$
  - Memory required?
    - $O(|V|^3)$
    - But we only use the computations from the previous step (k-1). So, we can only store the last step computations → $O(|V|^2)$

```
FLOYD-WARSHALL(W, n)
D^(0) ← W
for k ← 1 to n
    do for i ← 1 to n
           do for j ← 1 to n
                  do d_ij^(k) ← min(d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1))
return D^(n)
```

APSP: Floyd-Warshall

- Discussion
  - Note, if the given graph is sparse ($|E| \ll |V|^2$) and the weights are non-negative, then using Dijkstra's algorithm can be slightly more efficient.
    - $O(|V||E| \log |V|)$ with binary heap,
    - $O(|V|^2 \log|V| + |V||E|)$ with Fibonacci heap
  - Floyd-Warshall still has advantages:
    - Handles negative edges
    - Simple and straightforward implementation
    - No fancy data structures

Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components
- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

References

- The lecture slides are mainly based on the suggested textbooks and the corresponding published lecture notes:

  - CLRS: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.
  - KT: Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.
  - DPV: Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Algorithms, McGraw-Hill Higher Education., 2008.