Graph Algorithms: Single-Source Shortest Paths

Single-Source Shortest Paths

- **Given:** A single source vertex in a weighted, directed graph.
- Want to compute a shortest path for each possible destination.
  - Similar to BFS.
- We will assume either
  - no negative-weight edges, or
  - no reachable negative-weight cycles.
- Algorithm will compute a shortest-path tree.
  - Similar to BFS tree.

General Results

Theorem: Let $p = \langle v_0, v_2, \ldots, v_k \rangle$ be a SP from v_0 to v_k. Then, $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a SP from v_i to v_j, where $0 \le i \le j \le k$.

The proof can be seen using a cut-paste logic.
So, we have the optimal-substructure property.

Bellman-Ford's algorithm uses dynamic programming.

Dijkstra's algorithm uses the greedy approach.

Let $\delta(u, v)$ = weight of SP from u to v.

Corollary: Let p = SP from s to v, where $p = s \xrightarrow{p'} u \rightarrow v$.
Then, $\delta(s, v) = \delta(s, u) + w(u, v)$.

Corollary : Let $s \in V$. For all edges $(u, v) \in E$, we have
$\delta(s, v) \le \delta(s, u) + w(u, v)$.

Relaxation

Algorithms keep track of d[v], π[v]. **Initialized** as follows:

```
Initialize(G, s)
    for each v ∈ V[G] do
        d[v] := ∞;
        π[v] := NIL
    od;
    d[s] := 0
```

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

```
Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] := d[u] + w(u, v);
        π[v] := u
    fi
```

Bellman-Ford Algorithm

Can have negative-weight edges. Will "detect" reachable negative-weight cycles.

If there is a negative-weight cycle reachable from the source, no solution exists. Otherwise, produces shortest paths.

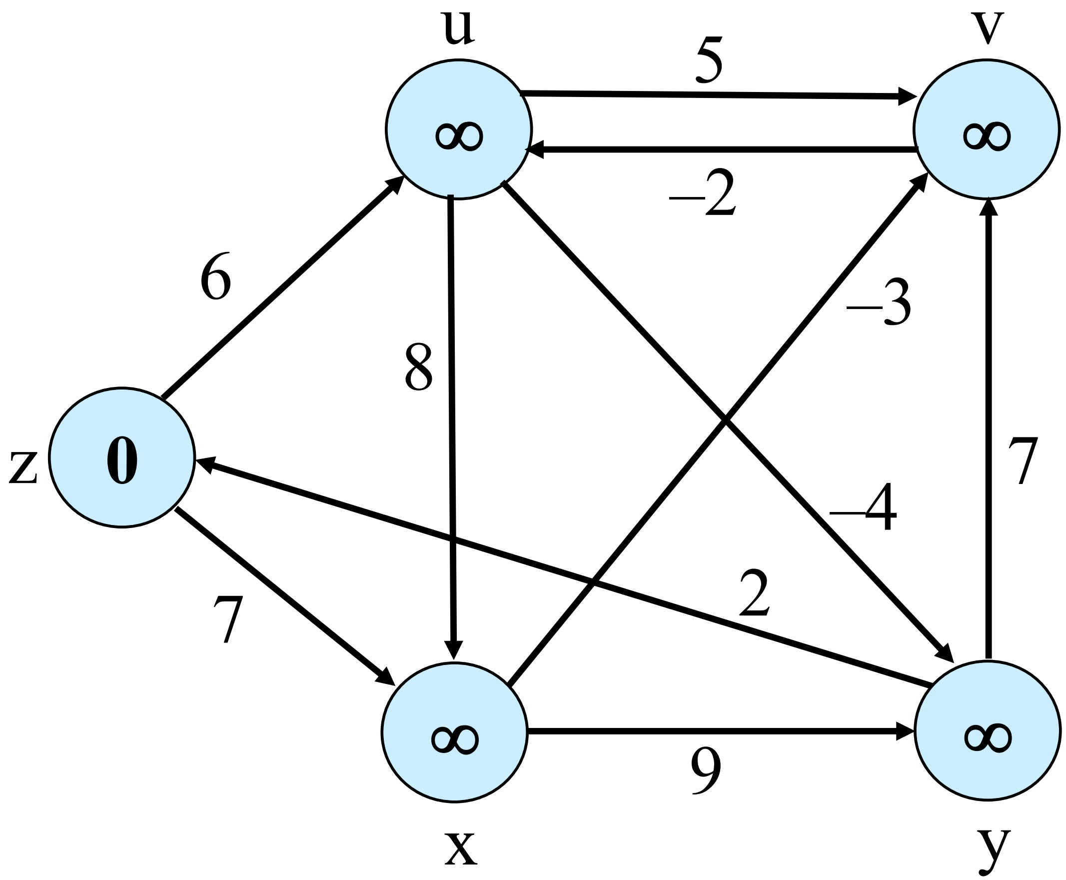
```
Initialize(G, s);
for i := 1 to |V[G]| –1 do
    for each (u, v) in E[G] do
        Relax(u, v, w)
    od
od;
for each (u, v) in E[G] do
    if d[v] > d[u] + w(u, v) then
        return false
    fi
od;
return true
```

Time Complexity?

$\boxed{O(VE)}$

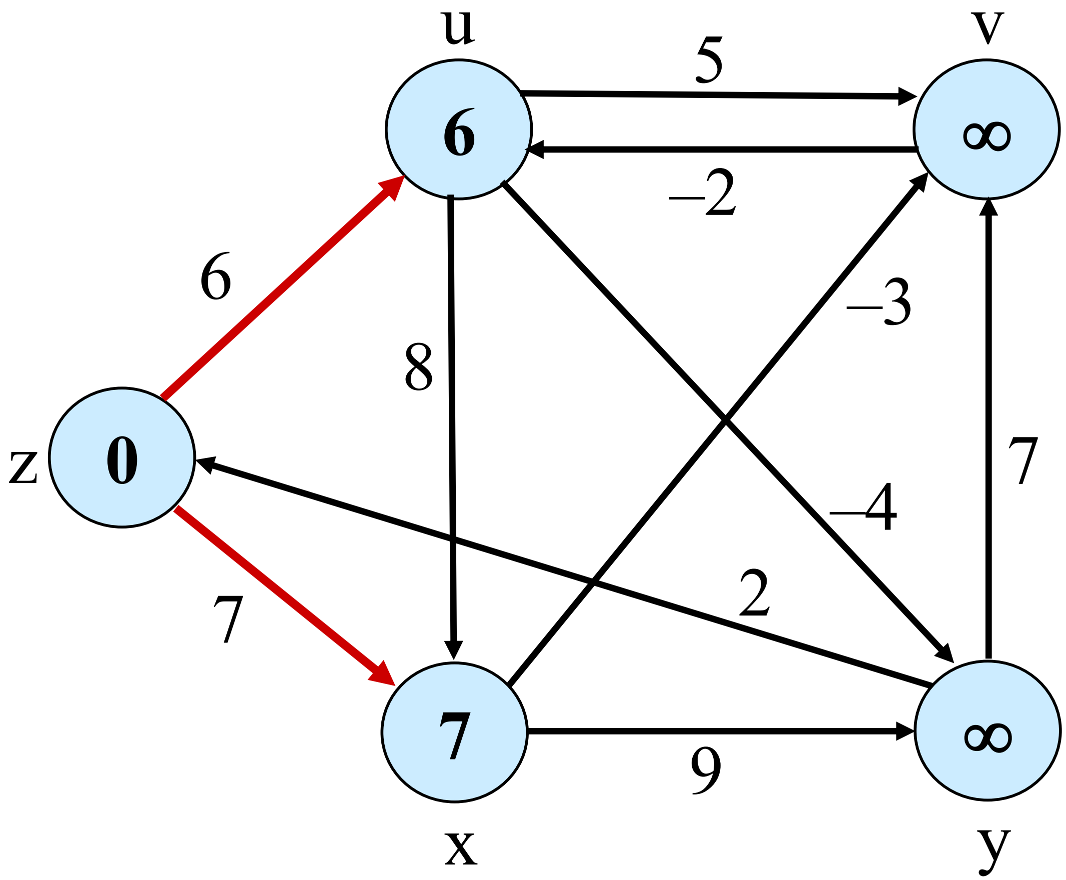
Example

```
Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] := d[u] + w(u, v);
        π[v] := u
    fi
```



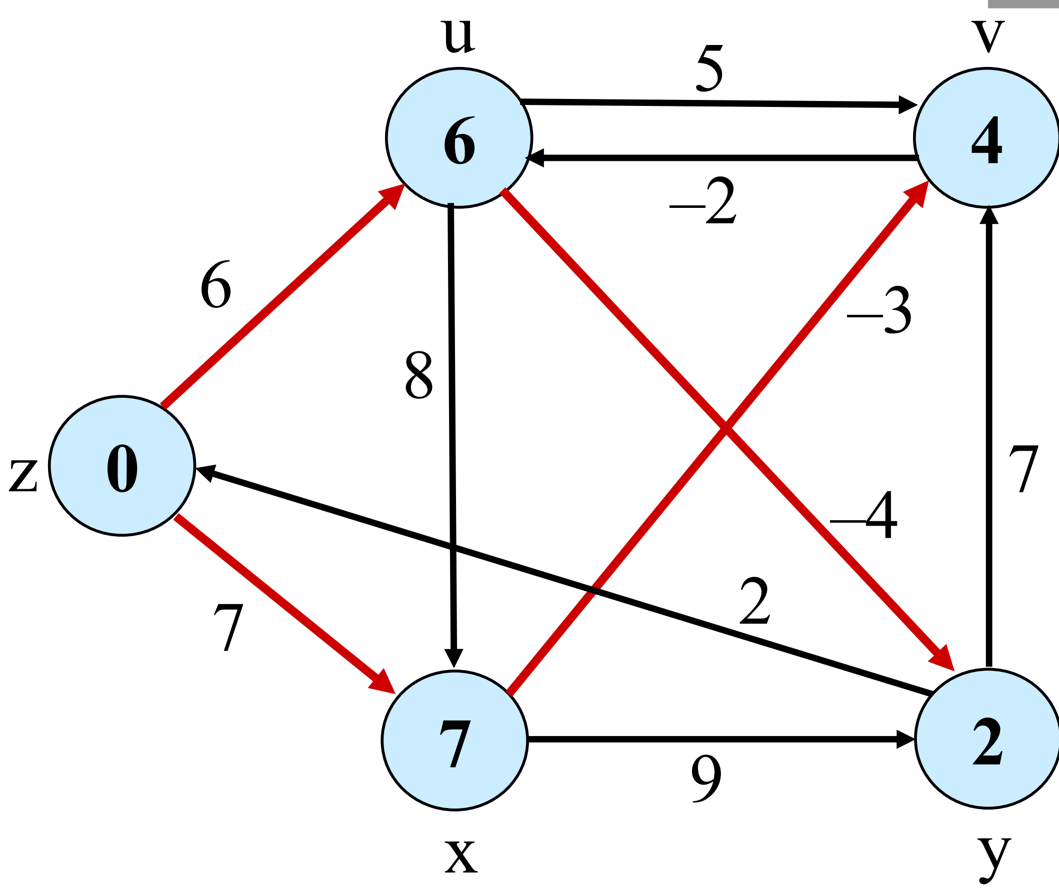
Example

Relax(u, v, w)
    if d[v] > d[u] + w(u, v) **then**
        d[v] := d[u] + w(u, v);
        π[v] := u
    fi



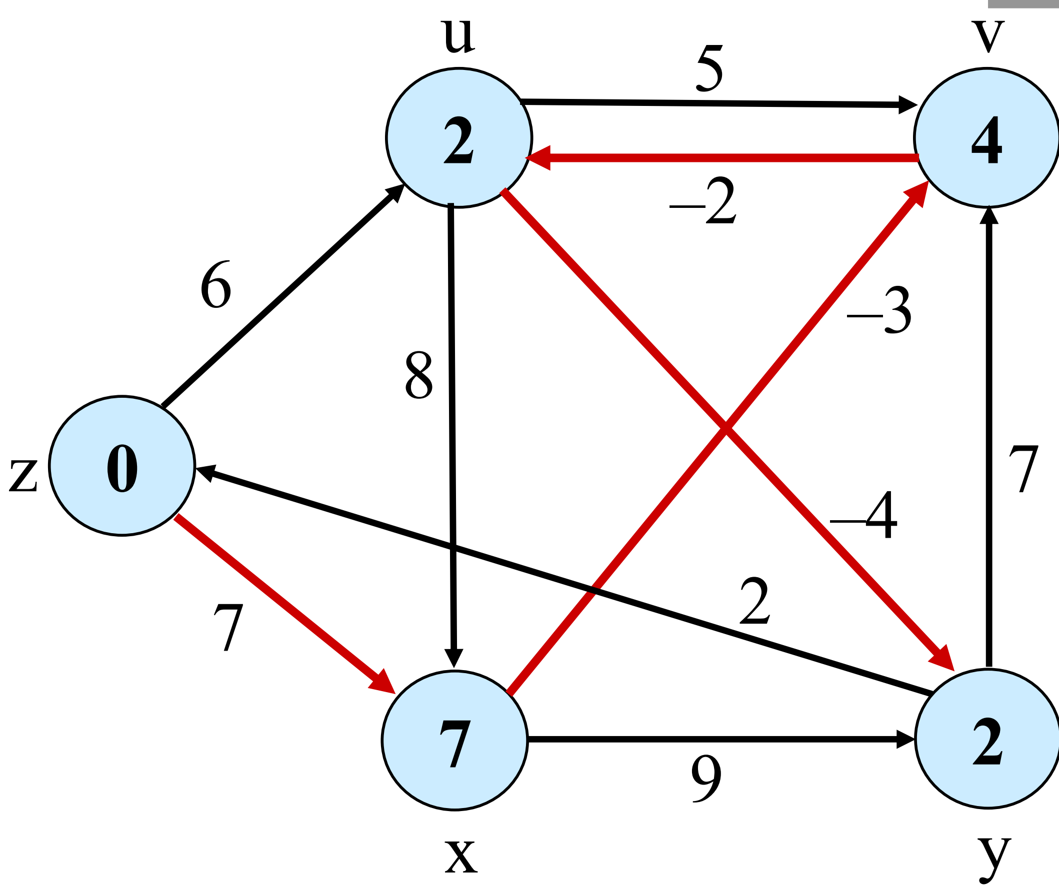
Example

Relax(u, v, w)
    if d[v] > d[u] + w(u, v) **then**
        d[v] := d[u] + w(u, v);
        π[v] := u
    fi



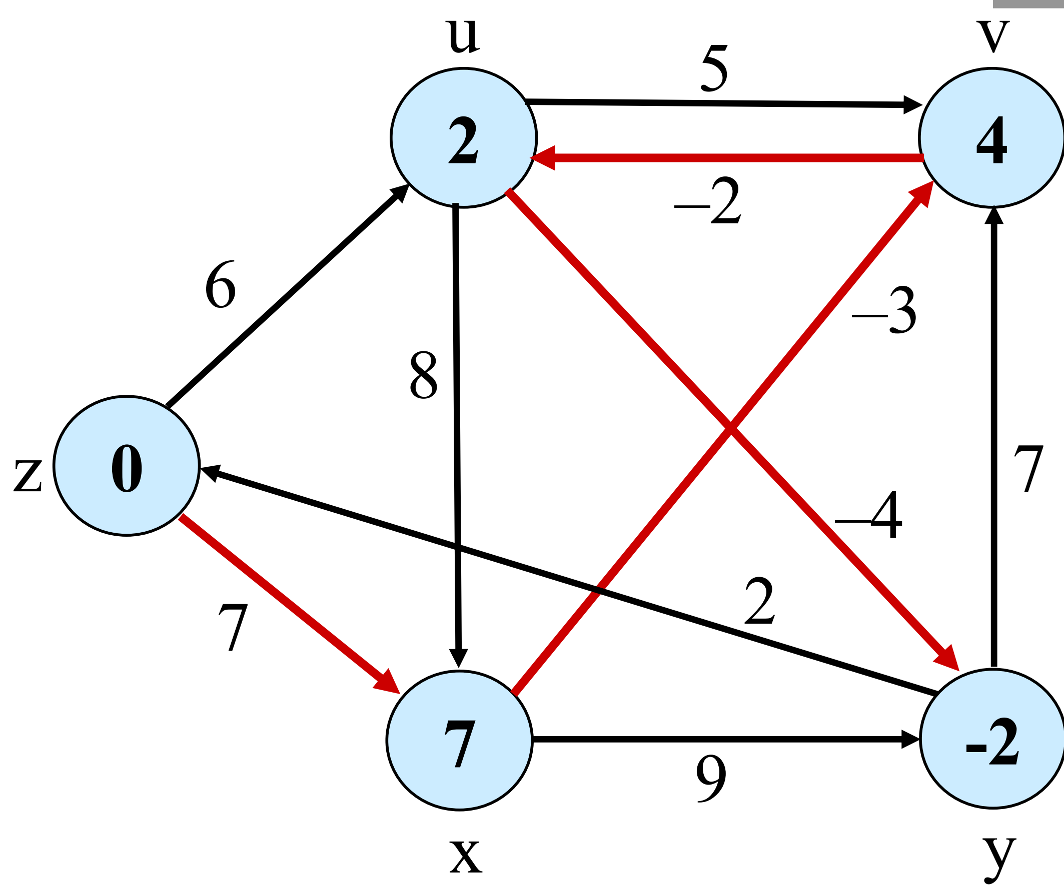
Example

Relax(u, v, w)
if d[v] > d[u] + w(u, v) **then**
d[v] := d[u] + w(u, v);
π[v] := u
fi



Example

```
Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] := d[u] + w(u, v);
        π[v] := u
    fi
```



If v is reachable through any shorter path through neighbor u, Relax(u, v, w) chooses that path from source to v→ correctness of the algorithm.

Dijkstra's Algorithm

Assumes **no negative-weight edges**.

Maintains a set S of vertices whose SP from s has been determined.

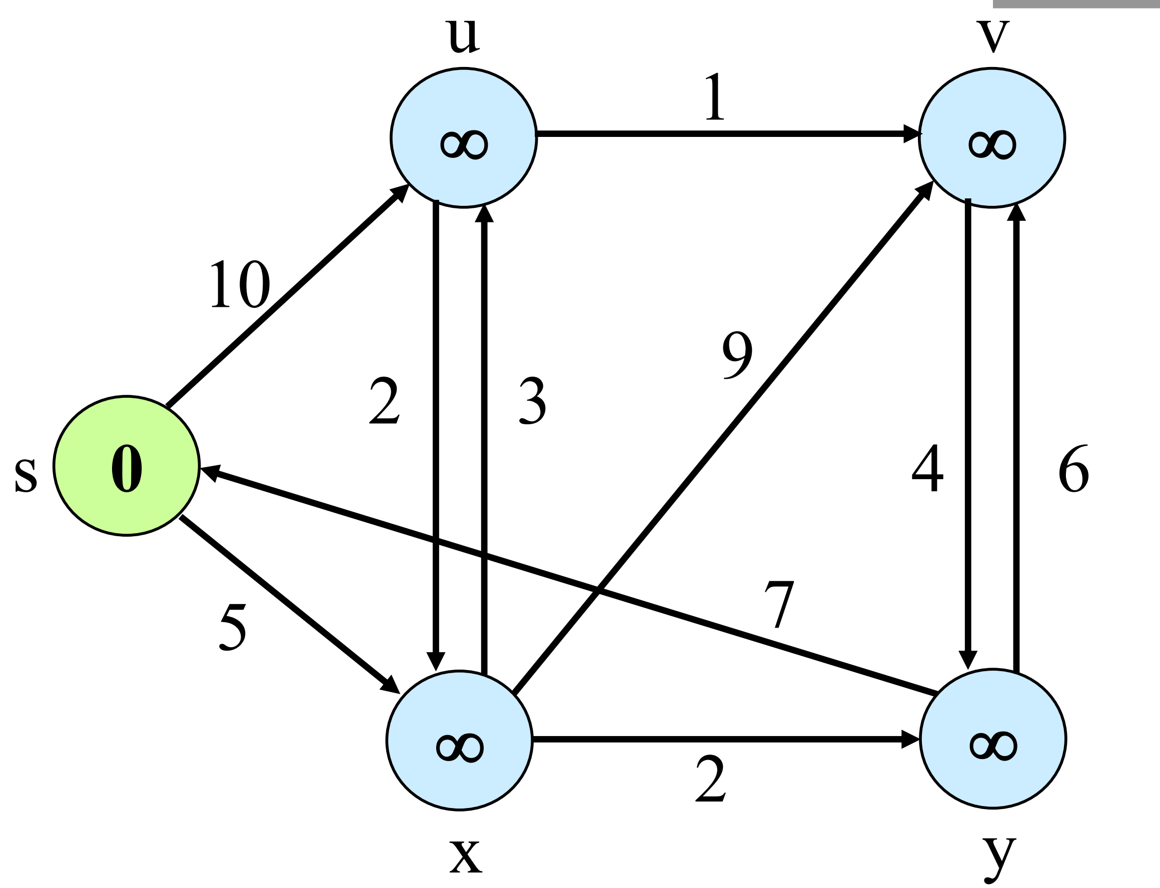
Repeatedly selects u in V–S with minimum SP estimate (greedy choice).

Store V–S in priority queue Q.

```
Initialize(G, s);
S := ∅;
Q := V[G];
while Q ≠ ∅ do
    u := Extract-Min(Q);
    S := S ∪ {u};
    for each v ∈ Adj[u] do
        Relax(u, v, w)
    od
od
```

Example

```
Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] := d[u] + w(u, v);
        π[v] := u
    fi
```

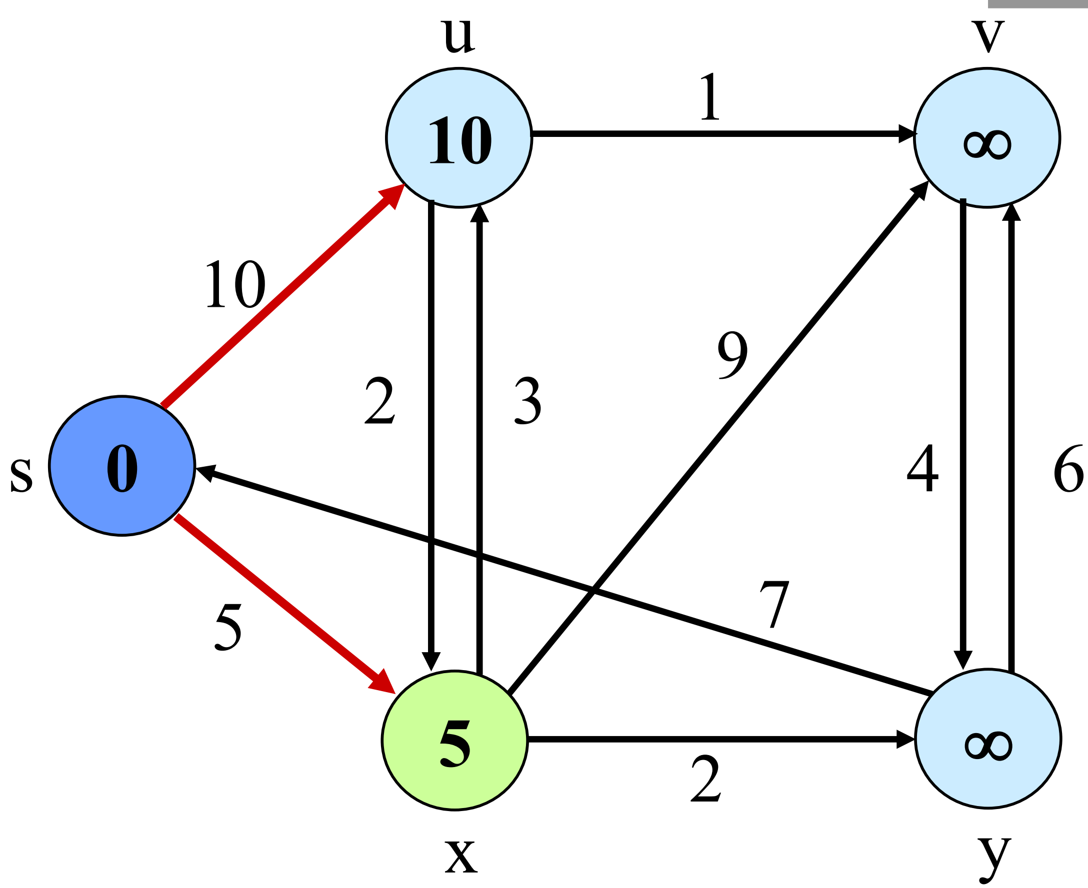


S={}

Q= V-S = V

Example

```
Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] := d[u] + w(u, v);
        π[v] := u
    fi
```

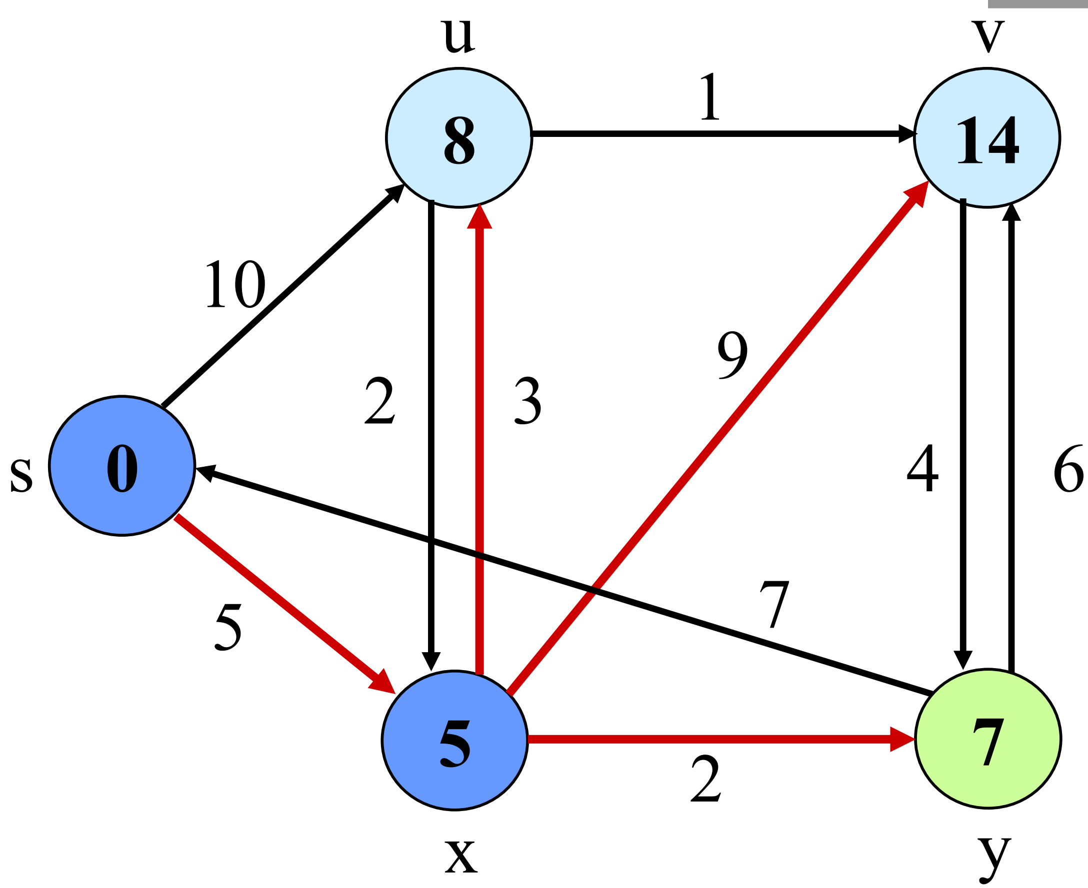


S={s}

Q= V-S = {u, v, x, y}

Example

```
Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] := d[u] + w(u, v);
        π[v] := u
    fi
```

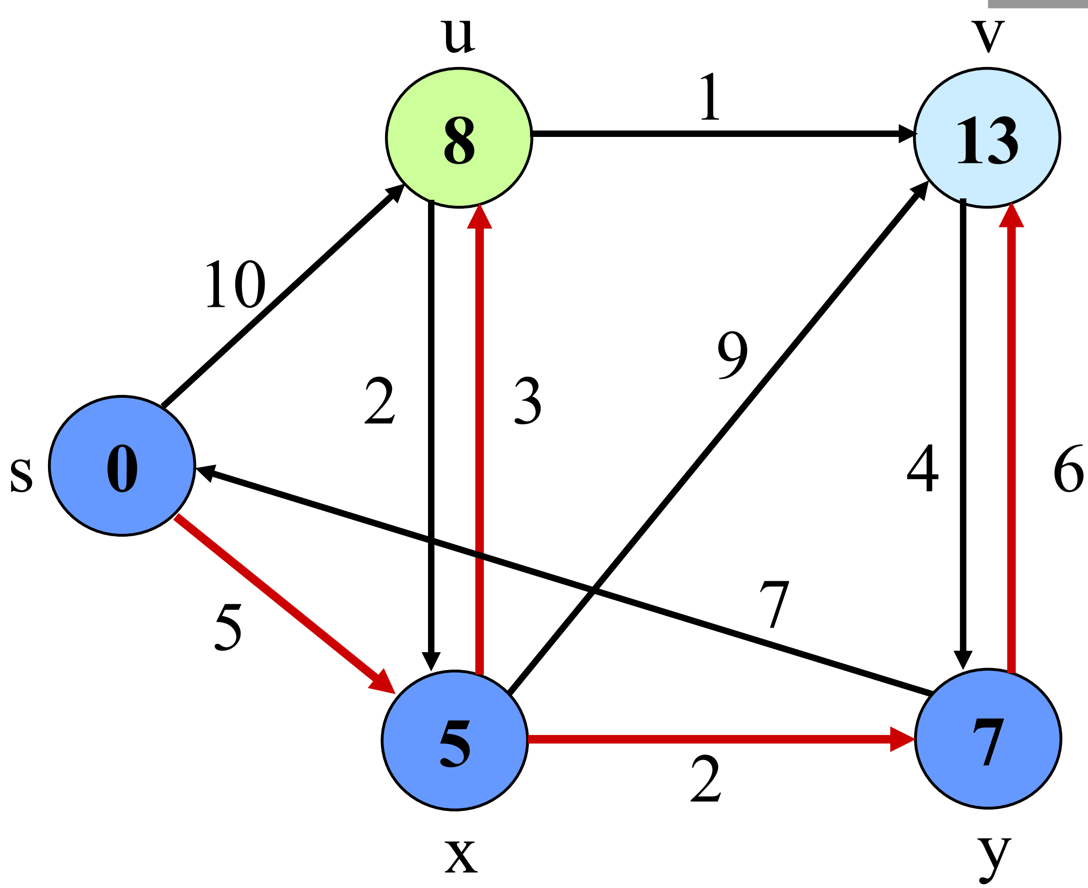


S={s, x}

Q= V-S = {u, v, y}

Example

```
Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] := d[u] + w(u, v);
        π[v] := u
    fi
```

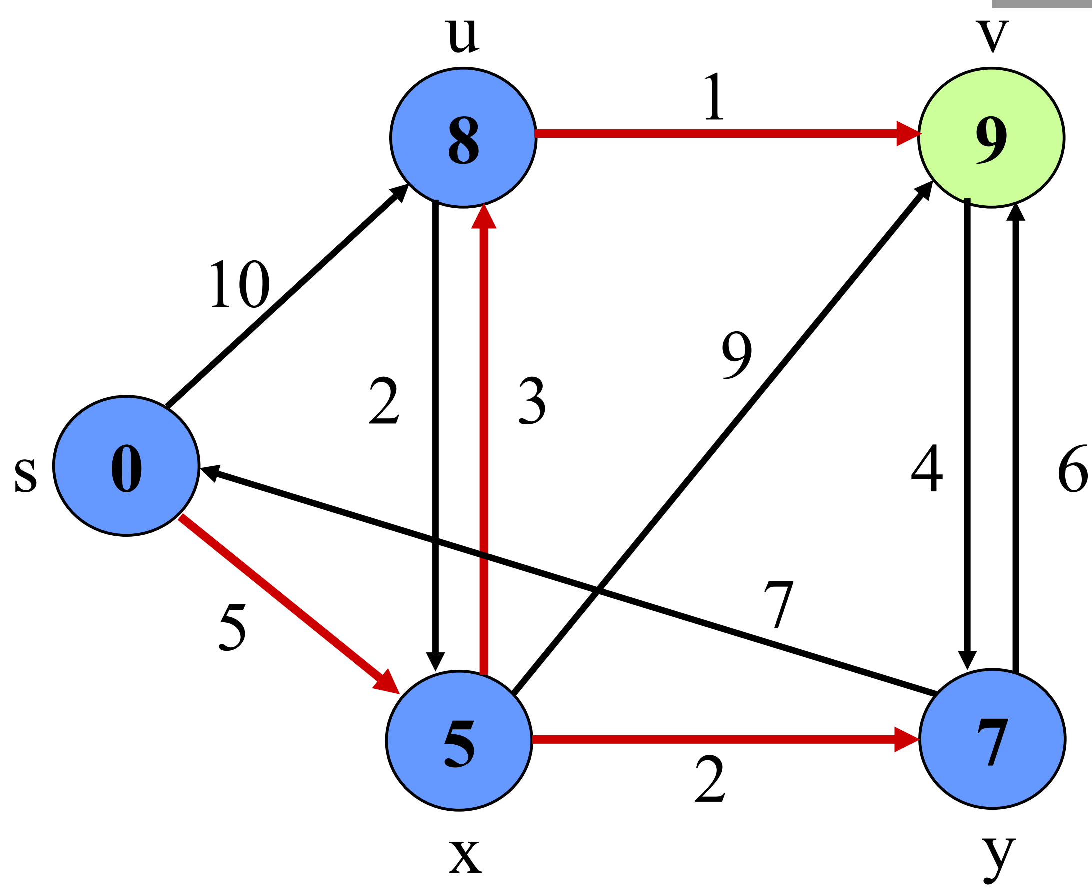


S={s, x, y}

Q= V-S = {u, v}

Example

Relax(u, v, w)
if d[v] > d[u] + w(u, v) **then**
d[v] := d[u] + w(u, v);
π[v] := u
fi

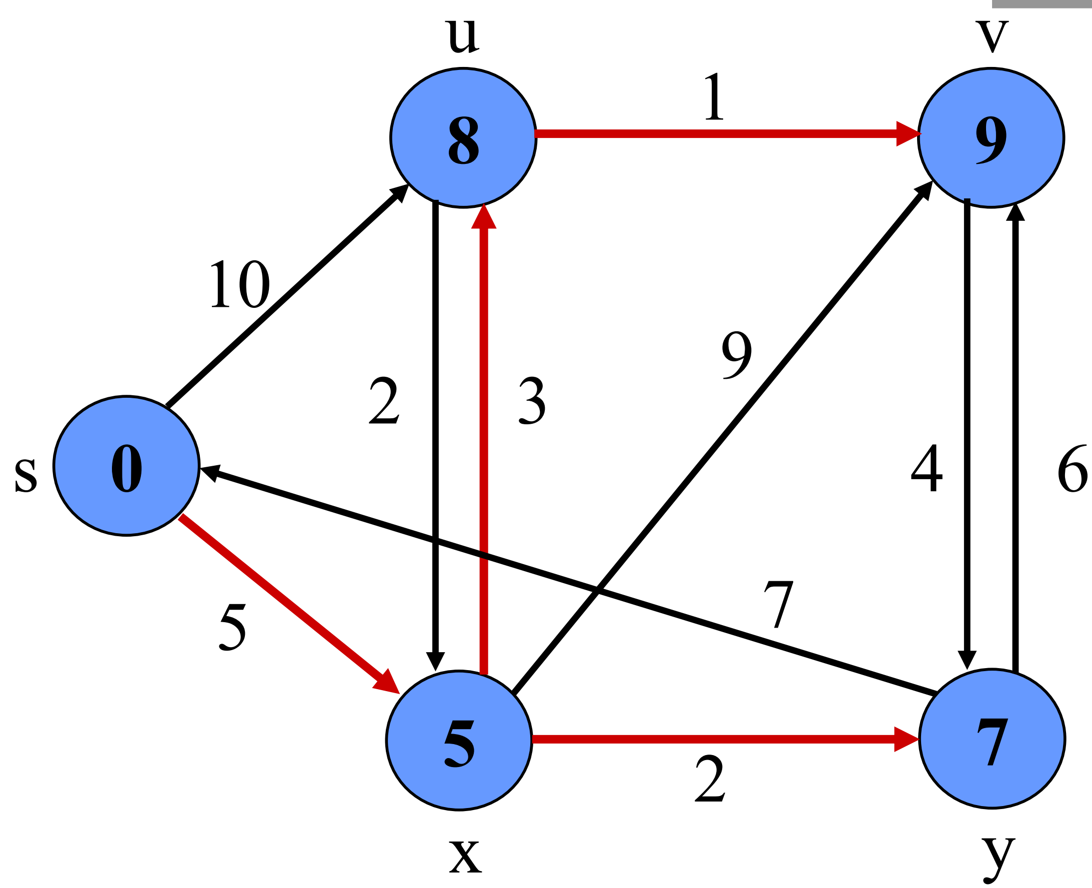


S={s, x, y, u}

Q= V-S = {v}

Example

Relax(u, v, w)
  if d[v] > d[u] + w(u, v) **then**
    d[v] := d[u] + w(u, v);
    π[v] := u
  fi



S={s, x, y, u, v}

Q= V-S = {}

Complexity

The analysis is similar to Prim's MST algorithm.

Running time is

$O(V^2)$ using linear array for priority queue.

$O((V + E) \lg V)$ using binary heap.

$O(V \lg V + E)$ using Fibonacci heap.

Difference from Prim's MST Alg.

In Prim's algorithm, we add the minimum cost edge to the total cost of current MST.

In contrast,

In Dijkstra, we add the cost cost(u,v) to the minimum path cost (from source) to u.

If there is a shorter path to v through neighbor u, Relax(u, v, w) chooses that path → correctness of the algorithm.