Object-oriented programming and data-structures

CS/ENGRD 2110 SUMMER 2018

Lecture 13: Shortest Path

http://courses.cs.cornell.edu/cs2110/2018su

Graph Algorithms

- Search
  - Depth-first search
  - Breadth-first search
- Shortest paths
  - Dijkstra's algorithm
- Spanning trees
  - Prim's algorithm
  - Kruskal's algorithm

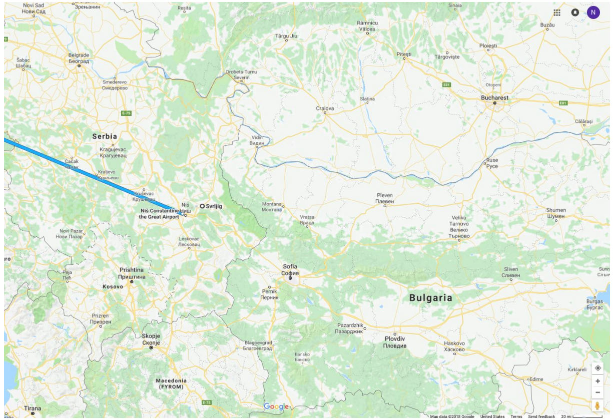
Shortest Path Problem

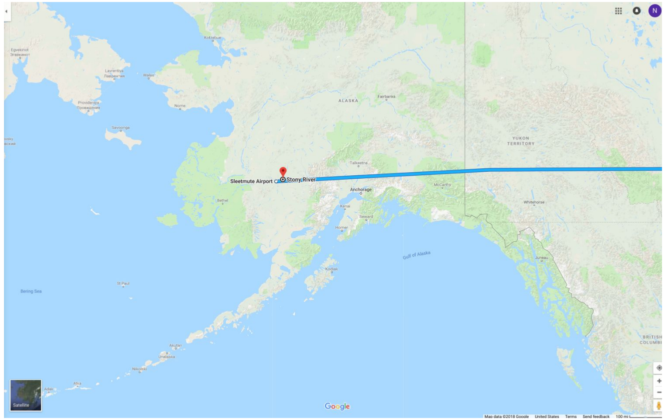
- How do I efficiently find the shortest path from **s** to **v** in a graph?

Shortest Path Problem

- How do I efficiently find the shortest path from **s** to **v** in a graph?

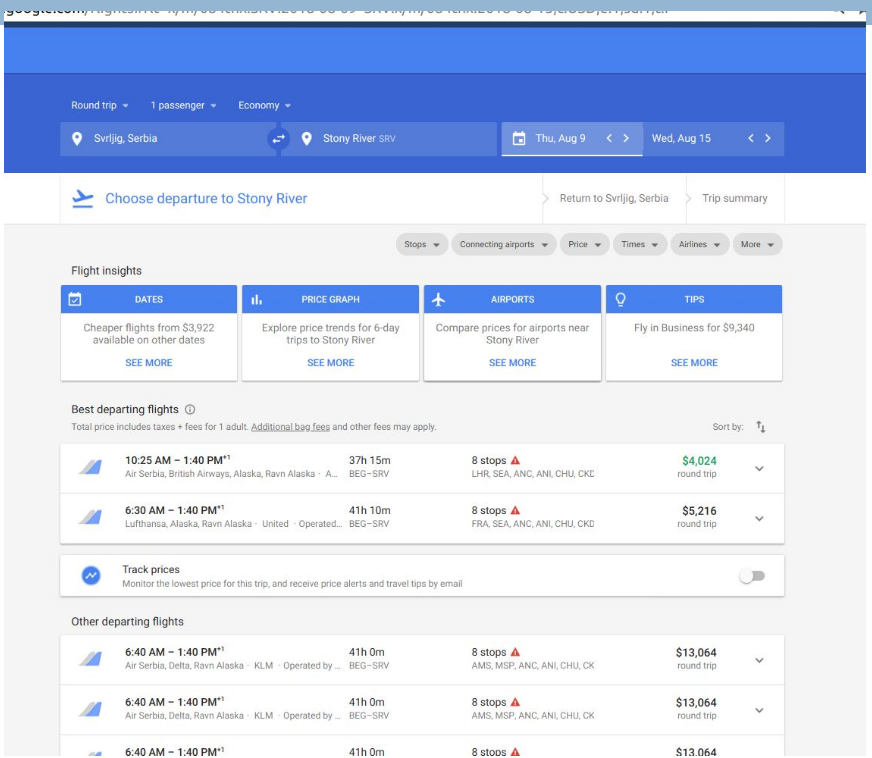
- What is the shortest path to fly from Svrljig (Serbia, Population: 7533) to Stony River (Alaska, USA, Population: 52)





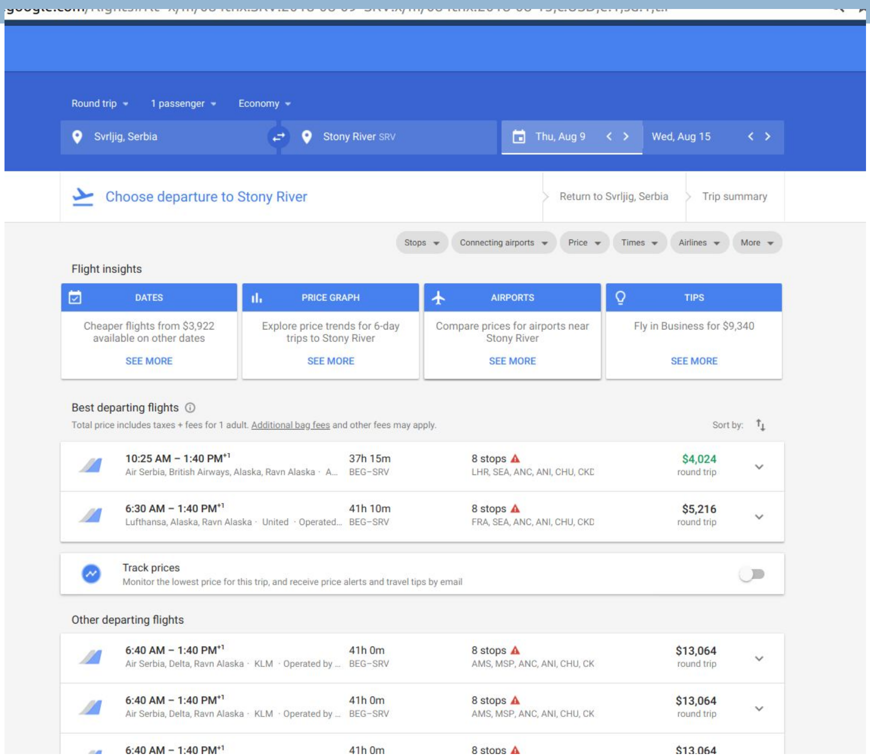
Shortest Path Problem

- Shortest path between Svrljig to Stony River requires **8 hops**



Shortest Path Problem

- Shortest path between Svrljig to Stony River requires **8 hops**
- Google Flights computed this is a few milliseconds. Billions of possible paths!
- Have we seen an algorithm that can compute the shortest path?



What about BFS

- BFS expands the graph in “layers”
  - First explores all nodes at distance 1 from the source
  - Next explores all nodes at distance 2 from the source, etc.

What about BFS

- BFS expands the graph in “layers”
  - First explores all nodes at distance 1 from the source
  - Next explores all nodes at distance 2 from the source, etc.

- But BFS only finds the path with the **smallest number of hops**

- Instead, we want to consider **weighted graphs**

Weighted Graphs

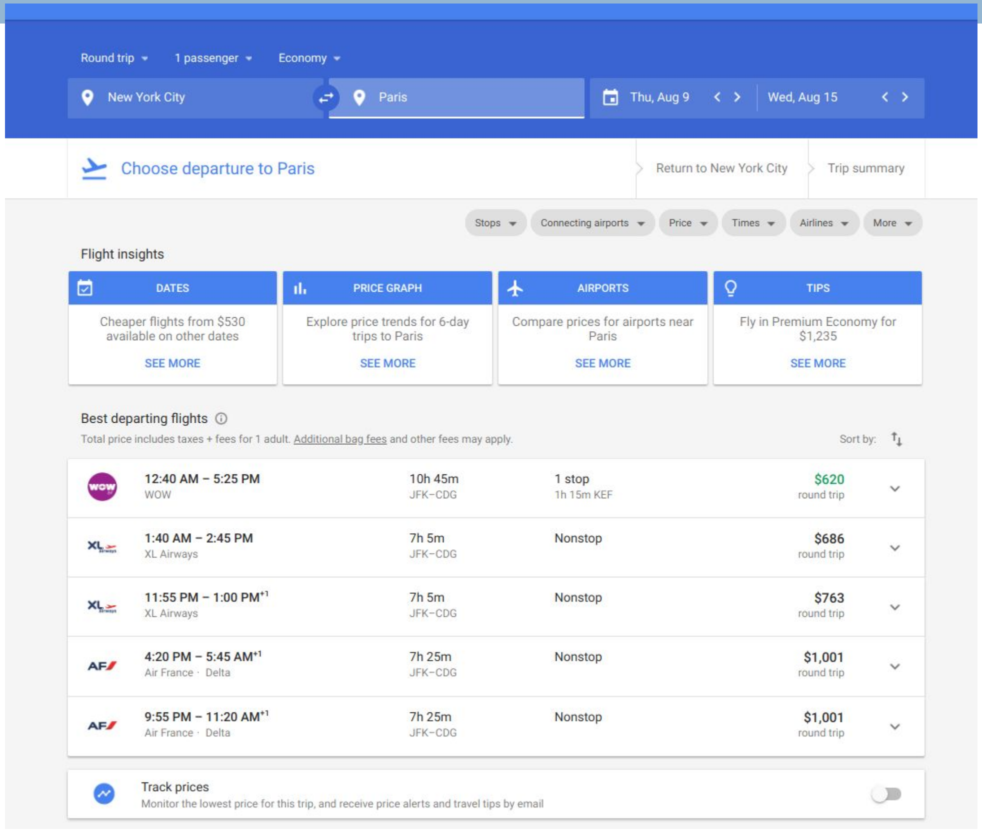
- In real graphs, want to assign **weights** to a graph
  - Price
  - Distance
  - Number of miles

- The shortest path is the **path with the lowest weight**, not necessarily the path with the smallest number of **edges**

Weighted Graphs

- In real graphs, want to assign **weights** to a graph
  - Price
  - Distance
  - Number of miles

- The shortest path is the **path with the lowest weight**, not necessarily the path with the smallest number of **edges**



Weighted Graphs, formally

- A weighted directed graph G = (V,E,W)
  - *V* is a (finite) set
  - *E* is a set of *ordered* pairs (*u*, *v*) where *u*,*v* ∈ *V*
  - *W is weight function that assigns edges to real-valued* ***weights***

Weighted Graphs, formally

- A weighted directed graph G = (V,E,W)
  - *V* is a (finite) set
  - *E* is a set of *ordered* pairs (*u*, *v*) where $u,v \in V$
  - *W is weight function that assigns edges to real-valued* ***weights***

- Recall that a path is a sequence of edges p = $(v_0, v_1, v_2, \ldots v_k)$
  - The weight w(p) of a path p = $(v_0, v_1, v_2, \ldots v_k)$ is the sum of the weights of its constituent edges
    - $$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Scoping the Problem

- Single Destination Shortest Paths Problem
  - Find a shortest path between two vertices **u and v**

Scoping the Problem

- Single Destination Shortest Paths Problem
  - Find a shortest path between two vertices **u and v**

- All-pairs shortest path problem
  - Find a shortest path from u to v for every pair of vertices u and v
    - Can run case-above for all vertices u and v
    - But exists a more efficient algorithm (Floyd-Warshall Algorithm)
    - **We do not look at this in this class!**

Single-Source Shortest Path (SSSP)

- Two algorithms:
  - Dijkstra's Algorithm
  - Bellman Ford Algorithm
- Dijkstra’s algorithm has complexity **O(V+E)**
- Bellman-Ford’s algorithm has complexity **O(VE)**
- Dijkstra works only for **positive edges.** Bellman-Ford works for both **positive and negative edges.**
- In this class we will only look at Dijkstra’s algorithm!

Single-Source Shortest Path (SSSP)

- Two algorithms:
  - Dijkstra's Algorithm
  - Bellman Ford Algorithm

Single-Source Shortest Path (SSSP)

- Two algorithms:
  - Dijkstra's Algorithm
  - Bellman Ford Algorithm

- Dijkstra’s algorithm has complexity **O(V+E)**

Single-Source Shortest Path (SSSP)

- Two algorithms:
  - Dijkstra's Algorithm
  - Bellman Ford Algorithm

- Dijkstra’s algorithm has complexity **O(V+E)**

- Bellman-Ford’s algorithm has complexity **O(VE)**

Single-Source Shortest Path (SSSP)

- Two algorithms:
  - Dijkstra's Algorithm
  - Bellman Ford Algorithm
- Dijkstra's algorithm has complexity **O((V+E)lgV)**
- Bellman-Ford's algorithm has complexity **O(VE)**
- Dijkstra works only for **positive edges.** Bellman-Ford works for both **positive and negative edges.**
- In this class we will only look at Dijkstra's algorithm!

Shortest Path - Definition

- We define the **shortest path weight** δ(u,v) from u to v by:

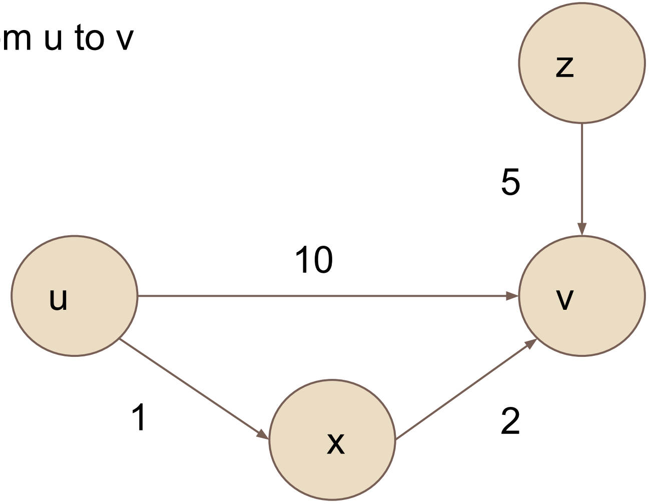
$$w(p) = \begin{cases} min(w(p) : u \rightsquigarrow v) & \text{If there is a path from u to v} \\ \infty & \text{Otherwise} \end{cases}$$

- A **shortest path** from vertex u to vertex v is then defined as any path **p** with weight **p = δ(u,v)**

Shortest Path - Definition

- We define the **shortest path weight** δ(u,v) from u to v by:

$$w(p) = \begin{cases} min(w(p) : u \leadsto v) & \text{If there is a path from u to v} \\ \infty & \text{Otherwise} \end{cases}$$



- A **shortest path** from vertex u to vertex v is then defined as any path **p** with weight **p = δ(u,v)**

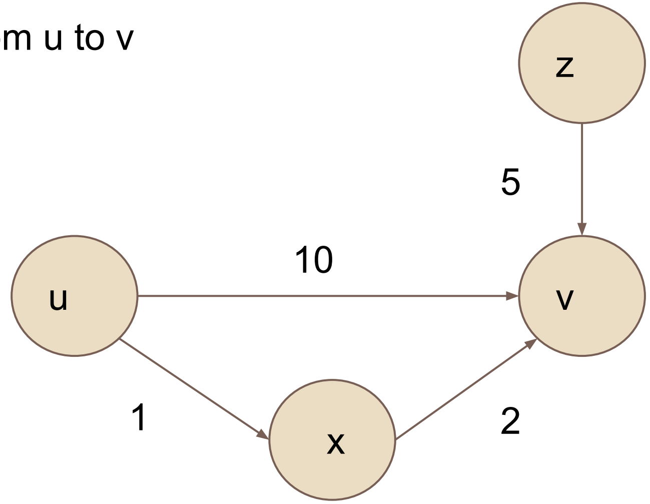
Shortest Path - Definition

- We define the **shortest path weight** δ(u,v) from u to v by:

$$w(p) = \begin{cases} min(w(p) : u \rightsquigarrow v) & \text{If there is a path from u to v} \\ \infty & \text{Otherwise} \end{cases}$$

- A **shortest path** from vertex u to vertex v is then defined as any path **p** with weight **p = δ(u,v)**

δ(u,v) = ?
δ(z,v) = ?
δ(z,u) = ?

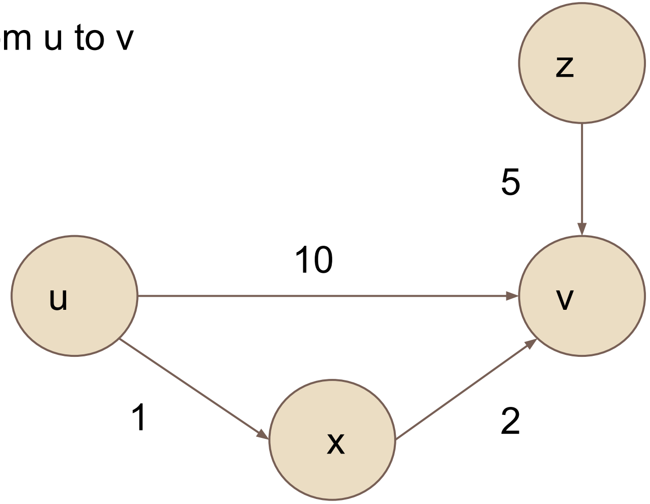


Shortest Path - Definition

- We define the **shortest path weight** δ(u,v) from u to v by:

$$w(p) = \begin{cases} min(w(p) : u \rightsquigarrow v) & \text{If there is a path from u to v} \\ \infty & \text{Otherwise} \end{cases}$$

δ(u,v) = 3
δ(z,v) = 5
δ(z,u) = ∞



- A **shortest path** from vertex u to vertex v is then defined as any path **p** with weight **p = δ(u,v)**

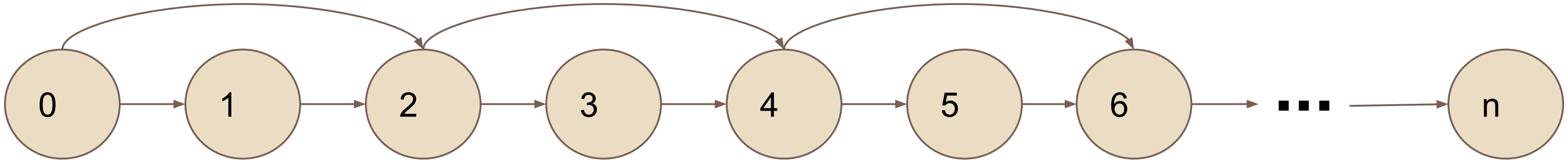
What about brute-force?

- What if we simply enumerated all paths between u and v, and picked the one with the smallest weight?
- How many paths between two nodes can there be in the worst-case?

What about brute-force?

- What if we simply enumerated all paths between u and v, and picked the one with the smallest weight?

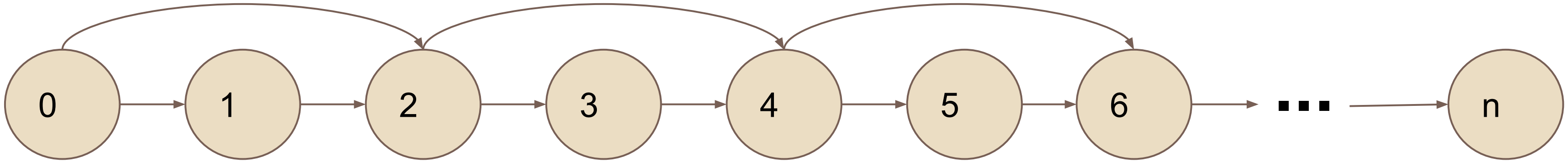
- How many paths between two nodes can there be in the worst-case?



Paths from 0 to 1?

What about brute-force?

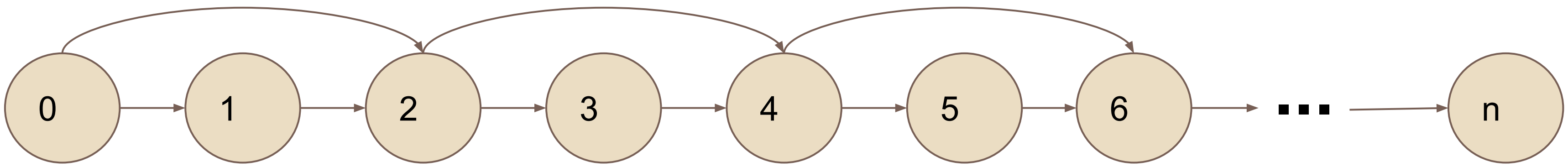
- What if we simply enumerated all paths between u and v, and picked the one with the smallest weight?
- How many paths between two nodes can there be in the worst-case?



Paths from 0 to 1? 1

What about brute-force?

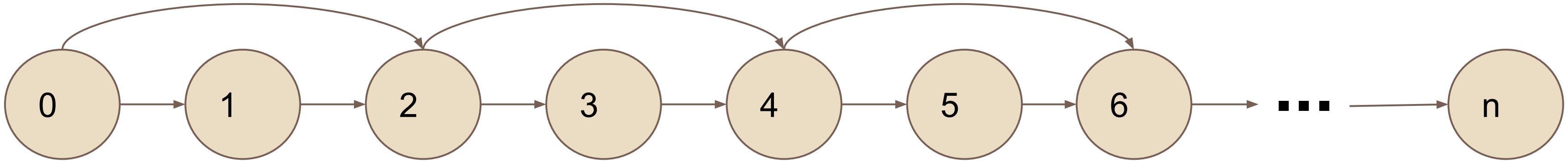
- What if we simply enumerated all paths between u and v, and picked the one with the smallest weight?
- How many paths between two nodes can there be in the worst-case?



Paths from 0 to 1? 1
Paths from 0 to 2?

What about brute-force?

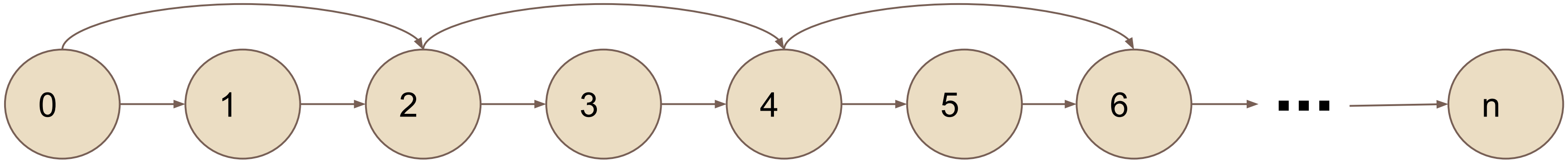
- What if we simply enumerated all paths between u and v, and picked the one with the smallest weight?
- How many paths between two nodes can there be in the worst-case?



Paths from 0 to 1? 1
Paths from 0 to 2? 2

What about brute-force?

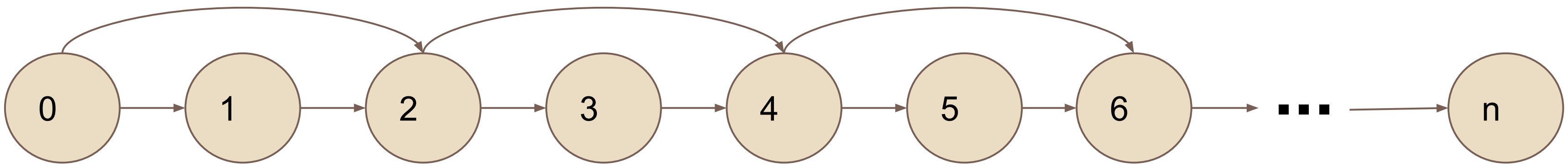
- What if we simply enumerated all paths between u and v, and picked the one with the smallest weight?
- How many paths between two nodes can there be in the worst-case?



Paths from 0 to 1? 1
Paths from 0 to 2? 2
Paths from 0 to 4?: 4

What about brute-force?

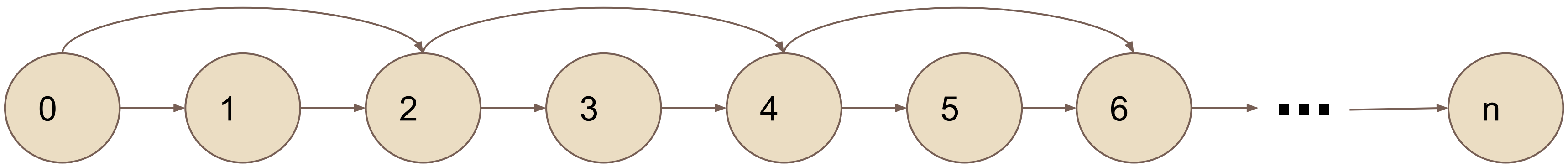
- What if we simply enumerated all paths between u and v, and picked the one with the smallest weight?
- How many paths between two nodes can there be in the worst-case?



Paths from 0 to 1? 1
Paths from 0 to 2? 2
Paths from 0 to 4?: 4
Paths from 0 to 6?: 8

What about brute-force?

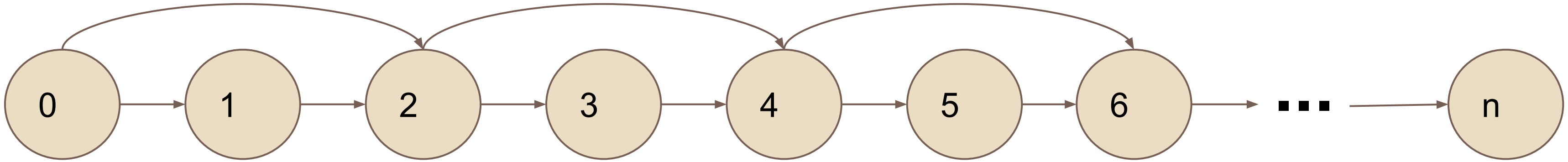
- What if we simply enumerated all paths between u and v, and picked the one with the smallest weight?
- How many paths between two nodes can there be in the worst-case?



Paths from 0 to 1? 1
Paths from 0 to 2? 2
Paths from 0 to 4?: 4
Paths from 0 to 6?: 8
Paths from 0 to 8? 16

What about brute-force?

- What if we simply enumerated all paths between u and v, and picked the one with the smallest weight?
- How many paths between two nodes can there be in the worst-case?

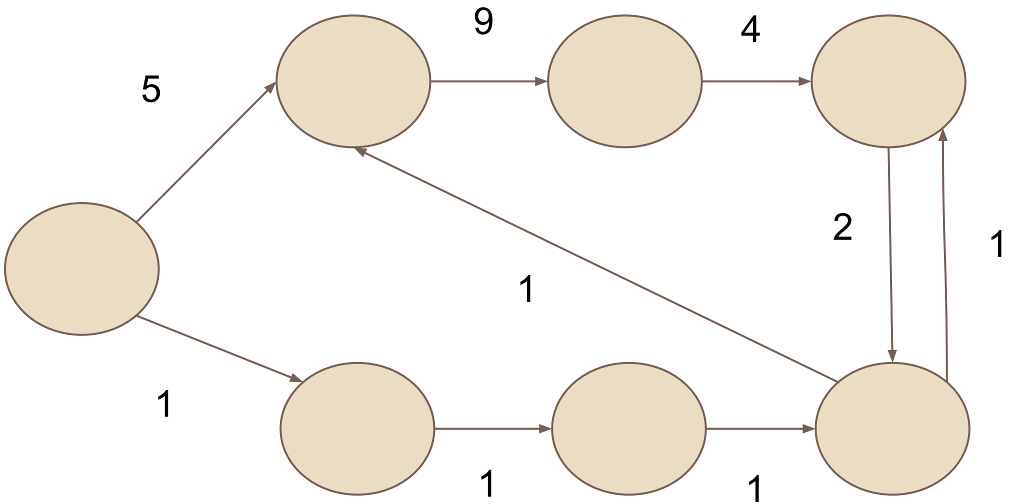


Paths from 0 to 1? 1
Paths from 0 to 2? 2
Paths from 0 to 4?: 4
Paths from 0 to 6?: 8
Paths from 0 to 8? 16

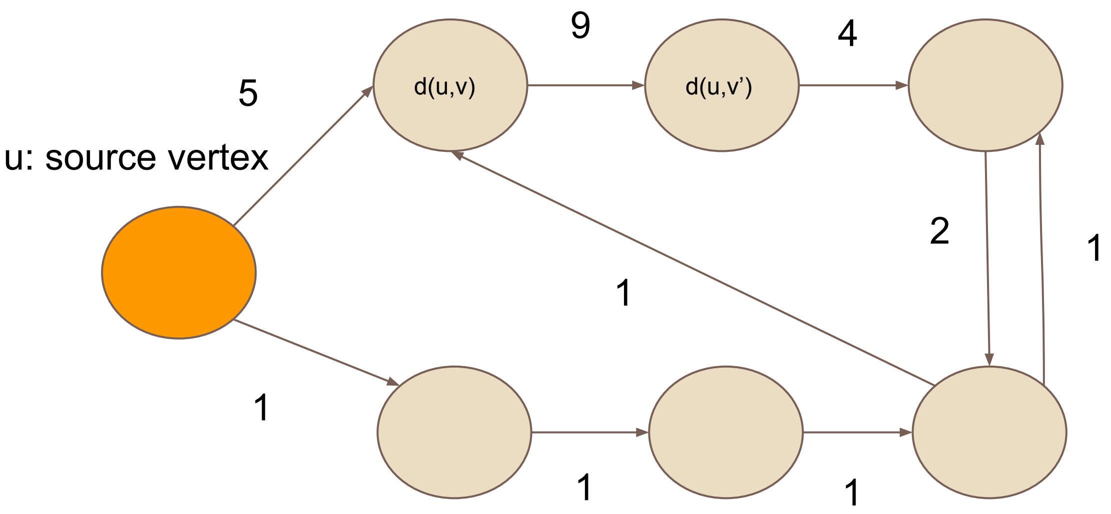
Order 2^(n/2)

Exponentially many paths

Terminology

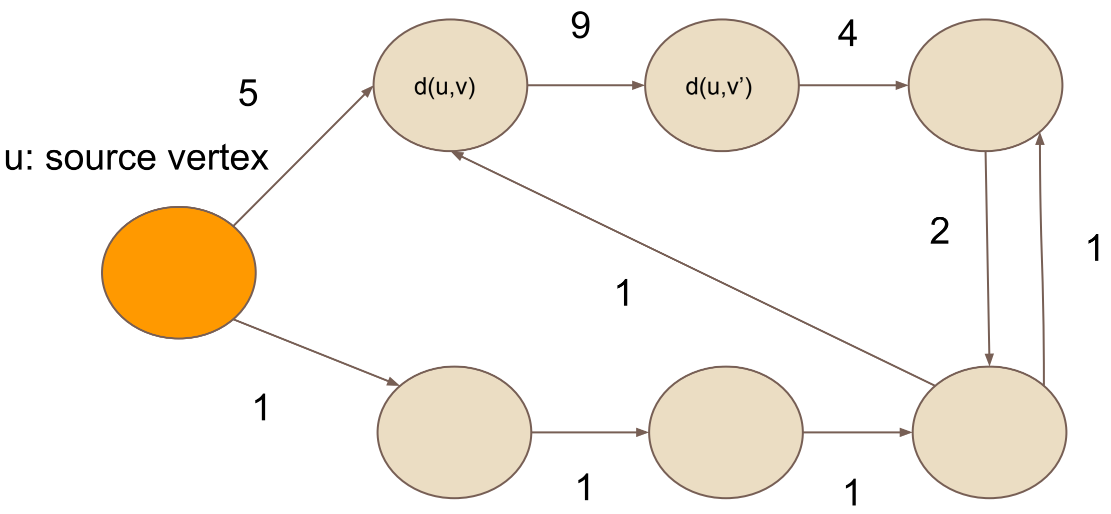


Terminology - Current Weight



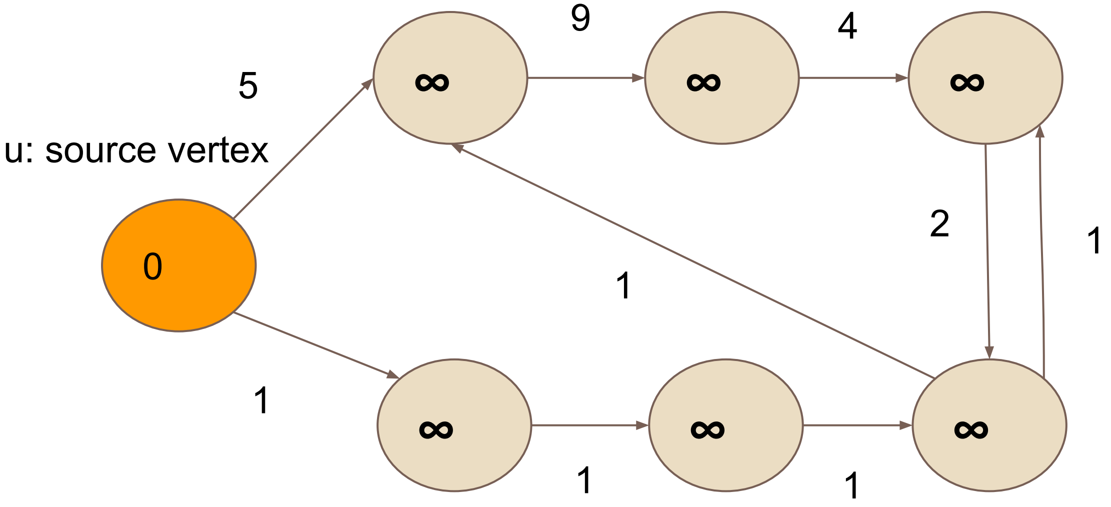
Write d(u,v) to be the **current weight** of node v: it represents the current best estimate of the shortest path from **u to v**

Terminology- Current Weight



Write d(u,v) to be the **current weight** of node v: it represents the current best estimate of the shortest path from **u to v**

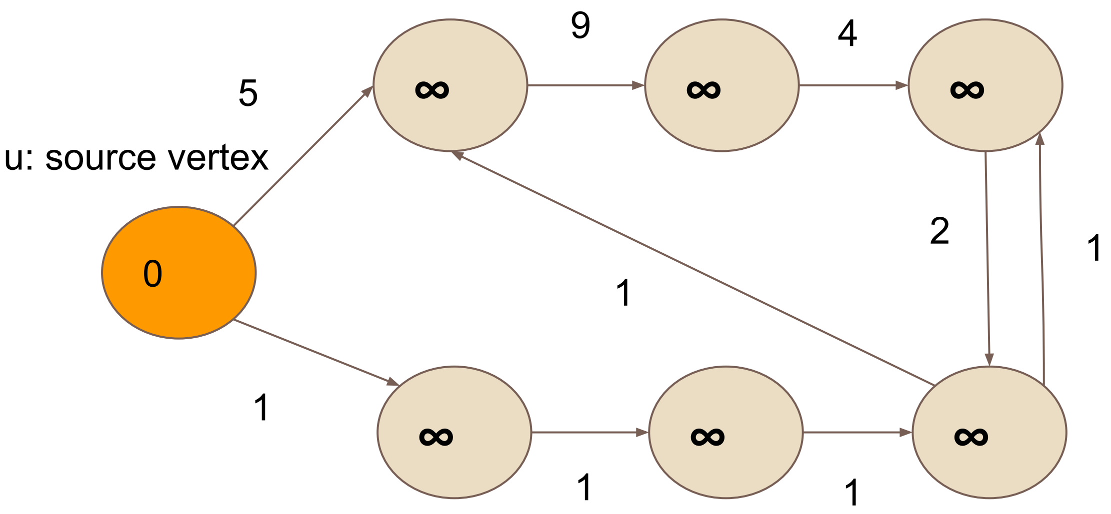
Terminology- Current Weight



Write d(u,v) to be the **current weight** of node v: it represents the current best estimate of the shortest path from **u to v**

Initially, because don't have an estimate, start with **∞**

Terminology- Current Weight

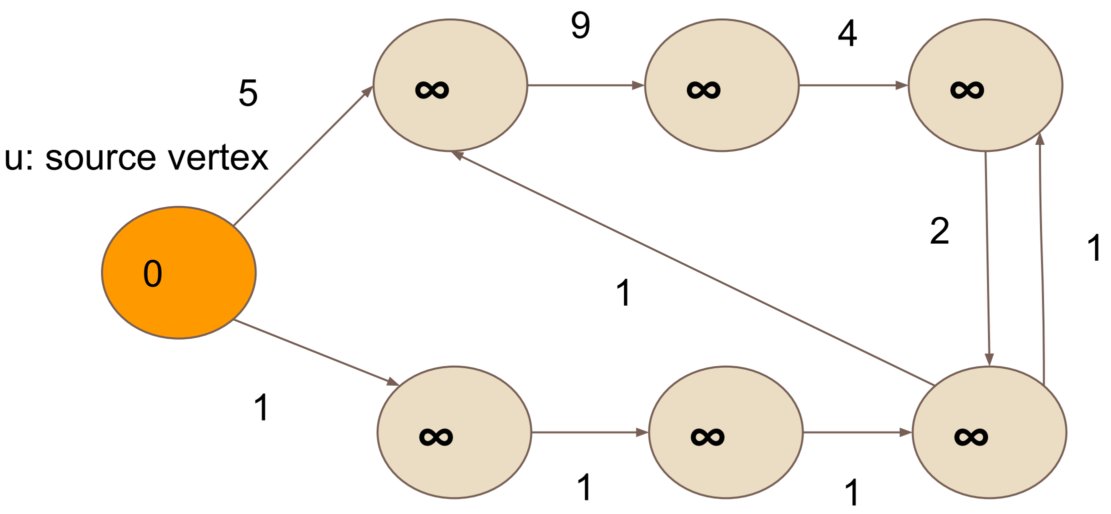


Write d(u,v) to be the **current weight** of node v: it represents the current best estimate of the shortest path from **u to v**

Initially, because don't have an estimate, start with **∞**

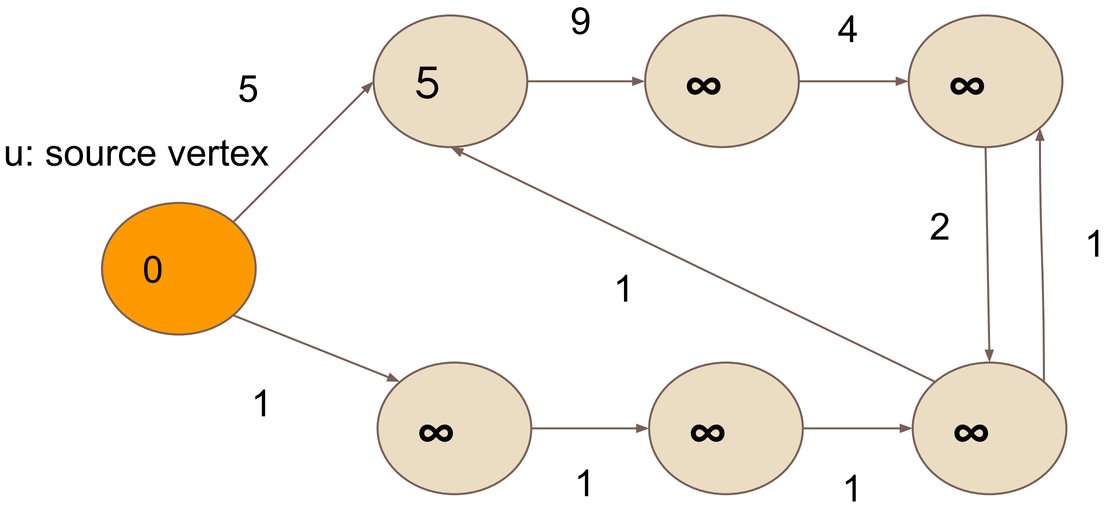
Goal: reduce d(u) until sure that d(u) = **δ(u,v)**

Terminology - Path Relaxation



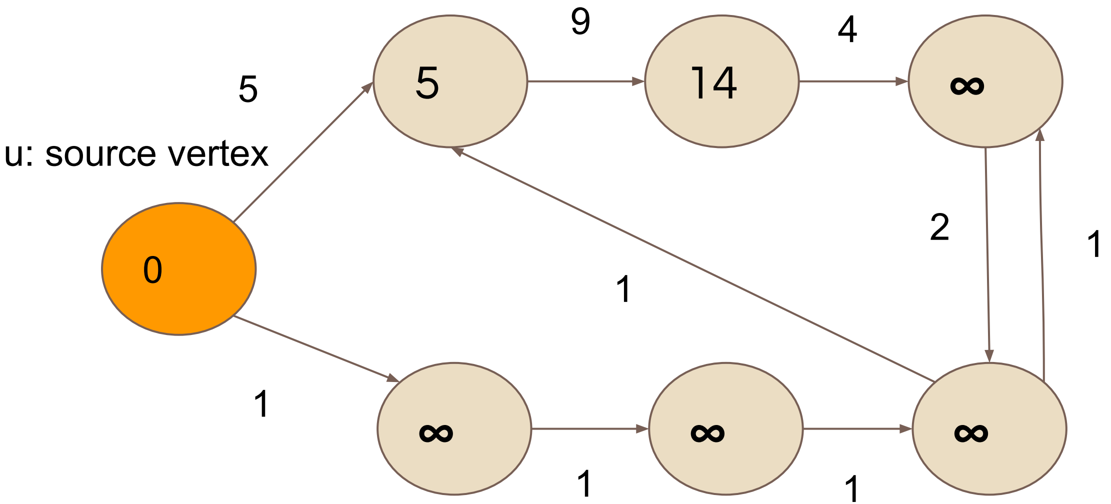
As discover new **paths**, will **update estimates** of what is currently the shortest path

Terminology - Path Relaxation



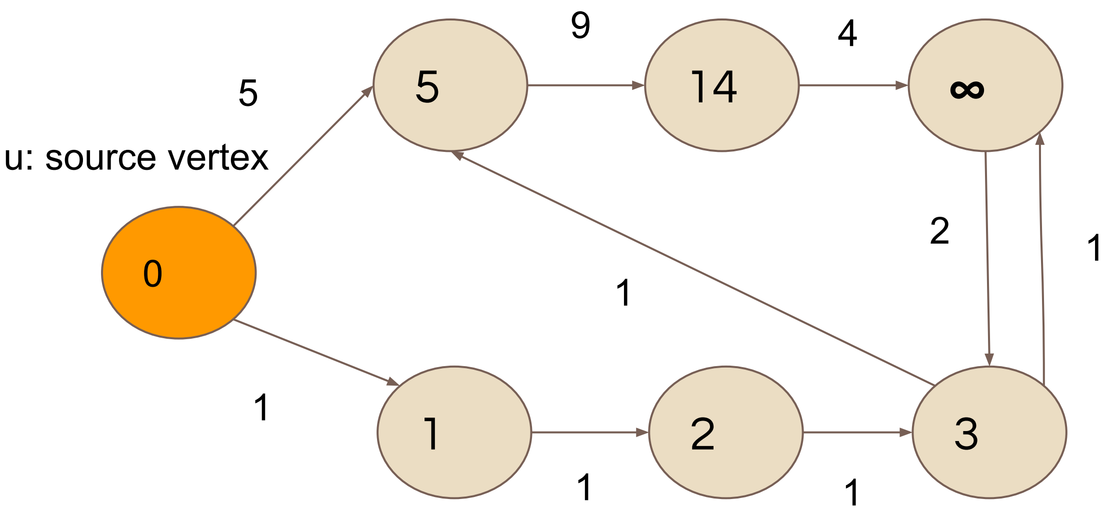
As discover new **paths**, will **update estimates** of what is currently the shortest path

Terminology - Path Relaxation



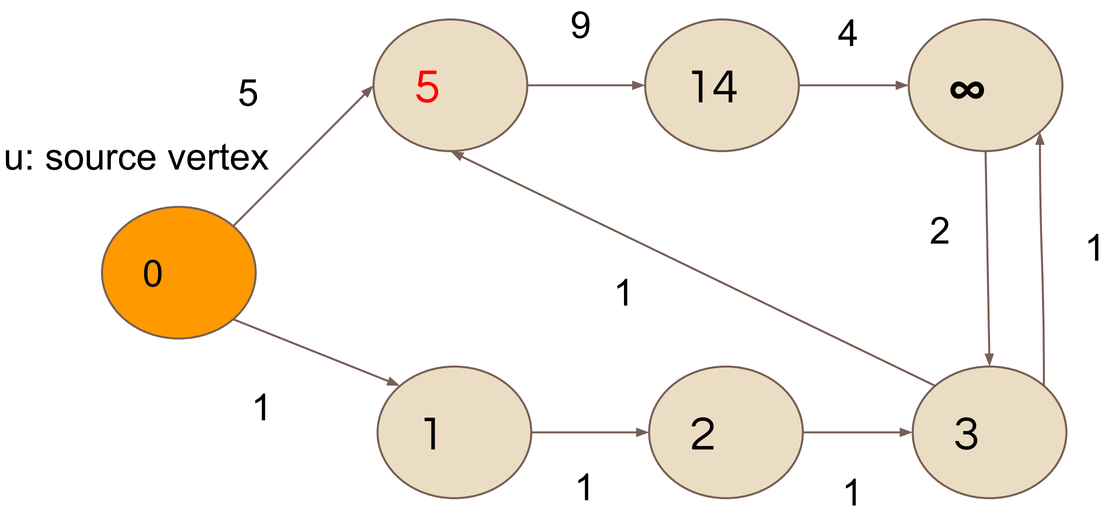
As discover new **paths**, will **update estimates** of what is currently the shortest path

Terminology - Path Relaxation



As discover new **paths**, will **update estimates** of what is currently the shortest path

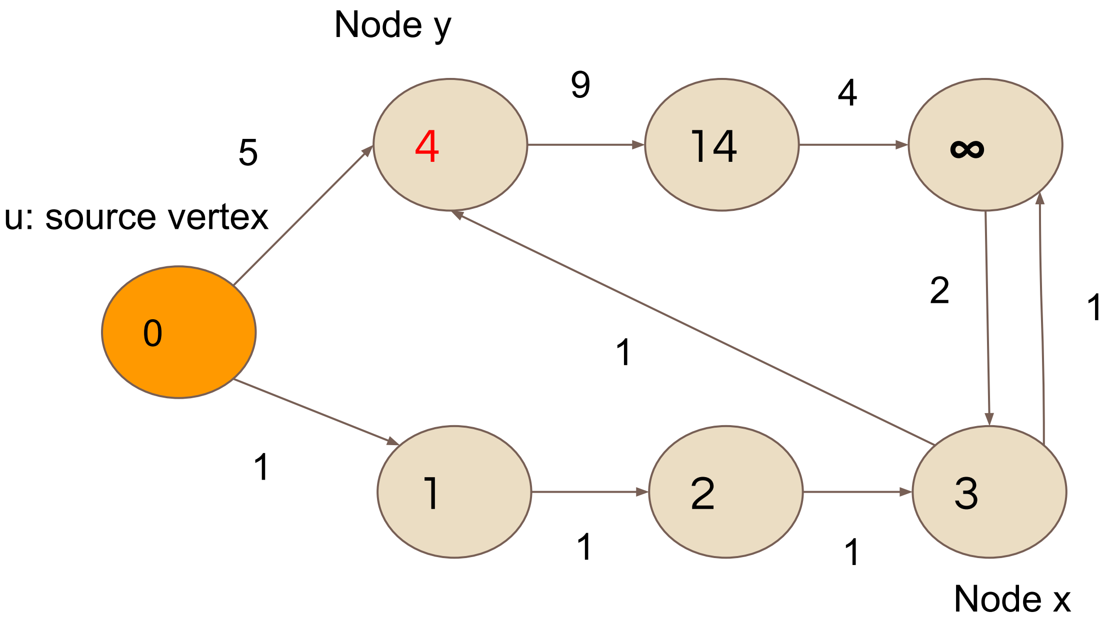
Terminology - Path Relaxation



Path relaxation:

Given a new edge (u,v):
If d[u] + w(u,v) < d[v], then we have discovered a better way to get from s to v, so update d[v] = d[u] + w(u,v)

Terminology - Predecessor



Keep track of the **predecessor of a node**: the node u that precedes v in the current estimate of the shortest path

Π[y] = x

Initially **Π[y] = null**

During path relaxation, if d[u] + w(u,v) < d[v], then update **Π[v] = u**

General Structure of SSSP

- **Initialisation**
  - For u in V: d[v] = ? **Π**[u] =?
  - d[s] = ?

General Structure of SSSP

- **Initialisation**
  - For u in V: d[v] = ∞ **Π**[u] = null
  - d[s] = 0

General Structure of SSSP

- **Initialisation**
  - For u in V: d[v] = ∞ Π[u] = null
  - d[s] = 0

- Repeat until **[When?]**
  - Select some edge (u,v) **[How?]**
    - Relax edge (u,v):
      - if d[v] > d[u] + w[u,v]
        - d[v] = d[u] + w[u,v]
        - Π[v] = u

General Structure of SSSP

- **Initialisation**
  - For u in V: d[v] = ∞ **Π**[u] = null
  - d[s] = 0

- Repeat until **none of the edges can be relaxed**
  - Select some edge (u,v) **[How?]**
    - Relax edge (u,v):
      - if d[v] > d[u] + w[u,v]
        - d[v] = d[u] + w[u,v]
        - **Π**[v] = u

General Structure of SSSP

- **Initialisation**
  - For u in V: d[v] = ∞ **Π**[u] = null
  - d[s] = 0

- Repeat until **none of the edges can be relaxed**
  - Select some edge (u,v) **[How?]**
    - Relax edge (u,v):
      - if d[v] > d[u] + w[u,v]
        - d[v] = d[u] + w[u,v]
        - **Π**[v] = u

Checking whether edges can be relaxed is O(E). Expensive!

General Structure of SSSP

- **Initialisation**
  - For u in V: d[v] = ∞ **Π**[u] = null
  - d[s] = 0

- Repeat until **none of the edges can be relaxed**
  - Select some edge (u,v) **[How?]**
    - Relax edge (u,v):
      - if d[v] > d[u] + w[u,v]
        - d[v] = d[u] + w[u,v]
        - **Π**[v] = u

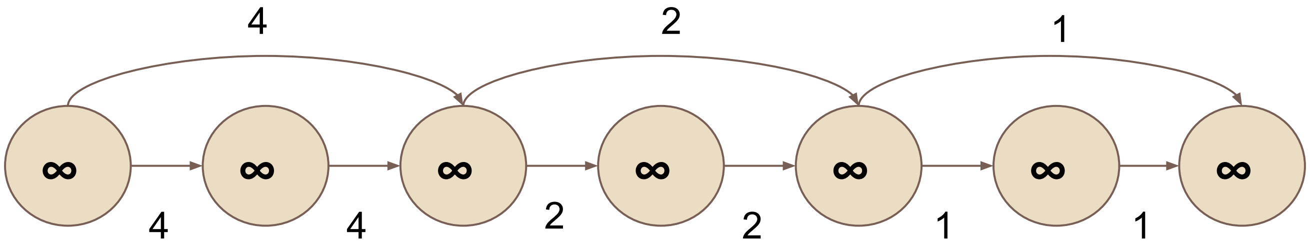
How many iterations will this do in the worst case?

General Structure of SSSP

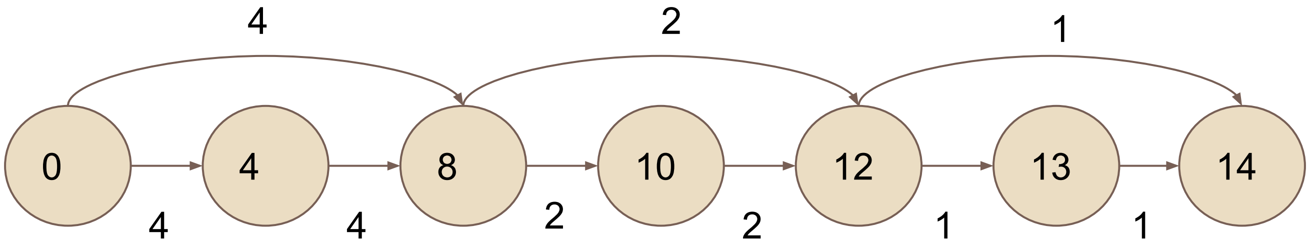
- **Initialisation**
  - For u in V: d[v] = ∞ **Π**[u] = null
  - d[s] = 0
- Repeat until **none of the edges can be relaxed**
  - Select some edge (u,v) **[How?]**
    - Relax edge (u,v):
      - if d[v] > d[u] + w[u,v]
        - d[v] = d[u] + w[u,v]
        - **Π**[v] = u

How many iterations will this do in the worst case?

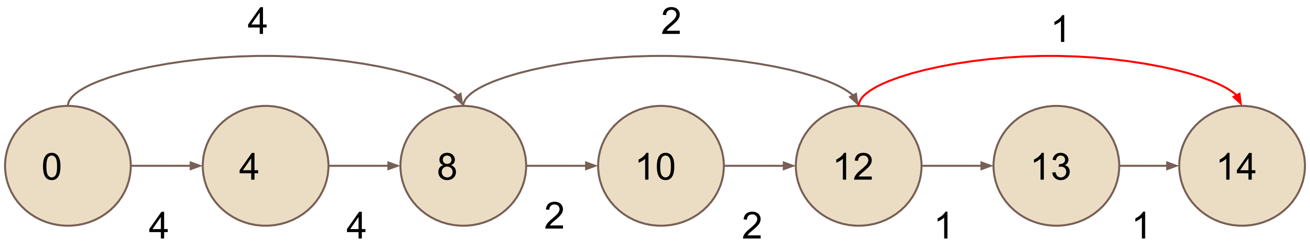
Worst-Case Iterations



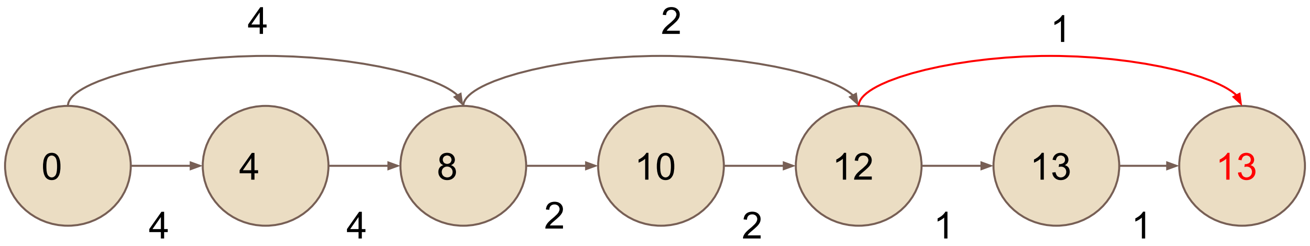
Worst-Case Iterations



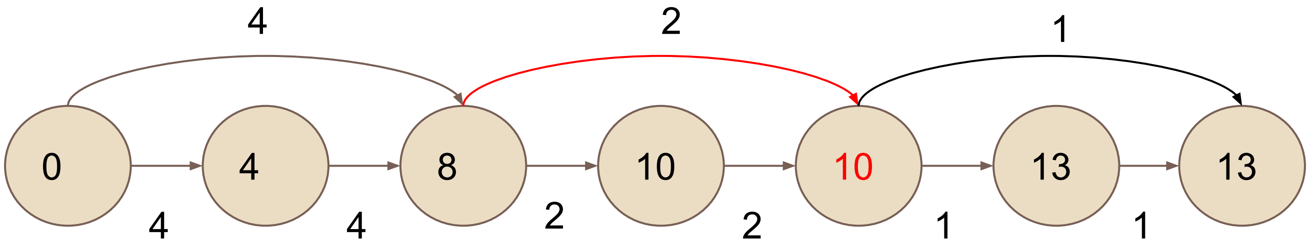
Worst-Case Iterations



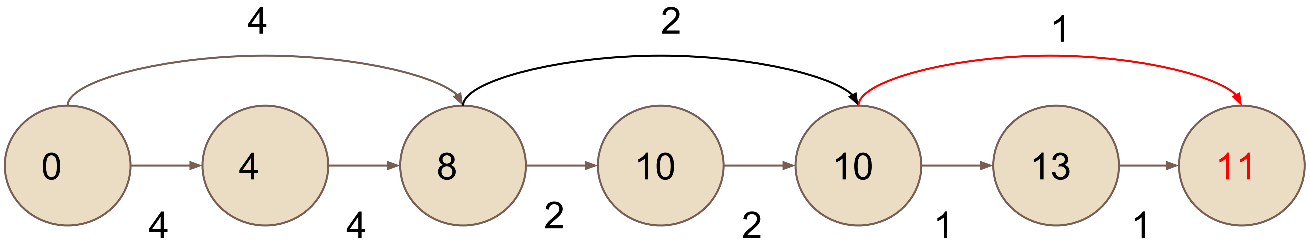
Worst-Case Iterations



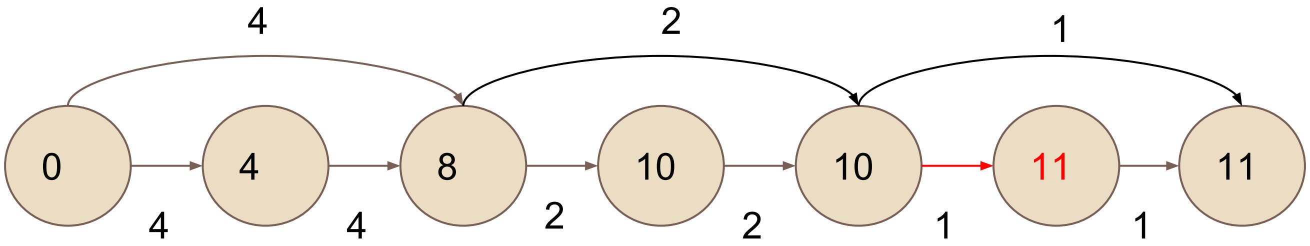
Worst-Case Iterations



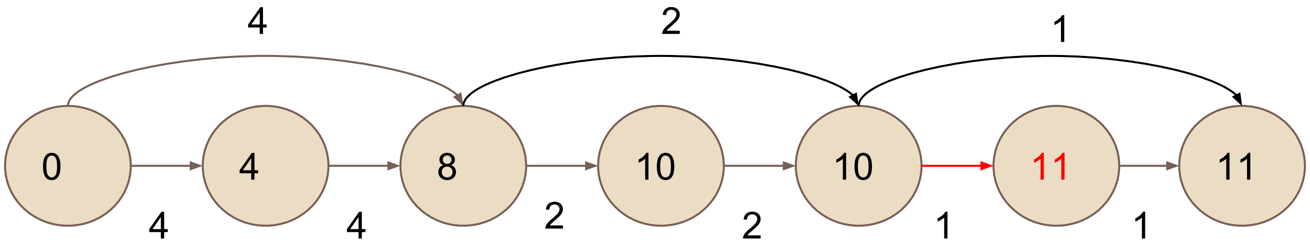
Worst-Case Iterations



Worst-Case Iterations



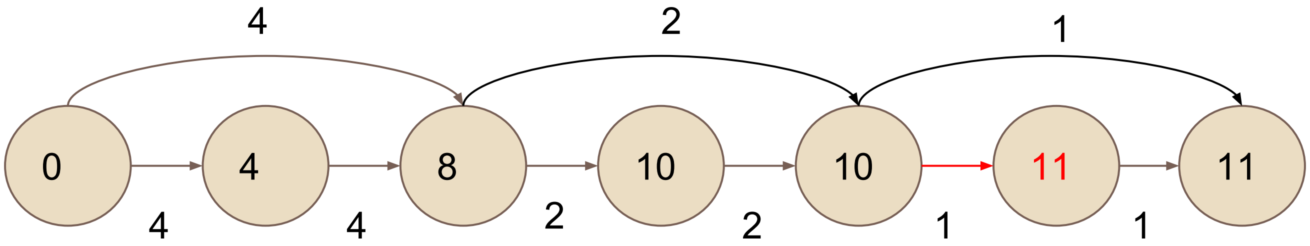
Worst-Case Iterations



Keep going decrementing from 13 (initial value), until shortest path value of 7

How many iterations does this take?

Worst-Case Iterations



Keep going decrementing from 13 (initial value), until shortest path value of 7

How many iterations does this take? 2^n/2 …

We have an exponential algorithm! (Again!)

Need to find some way to “intelligently” select the edges.

Dijkstra's algorithm

- We need a way to **bound** the number of times that we relax edges
- Dijkstra's algorithm does this by **greedily** selecting the vertex v with the smallest d(u,v) and **relaxing** its neighbouring edges.
- We'll see how this is sufficient to guarantee that d(u,v)=**δ**(u,v) once all vertices have been processed
- It only requires 1 pass on all the vertices (V) and all the edges (E)!
- The algorithm itself is surprisingly simple. The proof is harder.

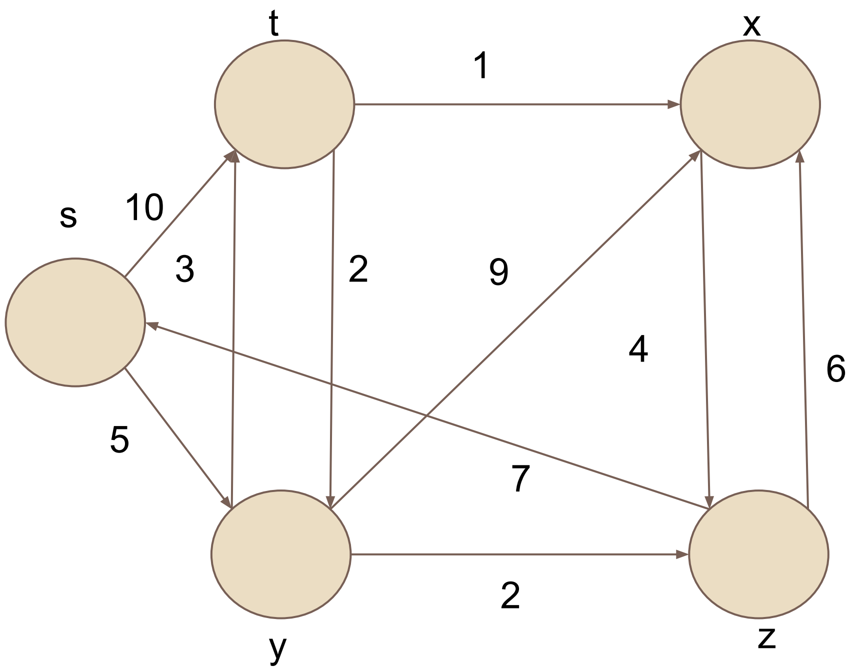
Dijkstra's algorithm

- Maintains a set **S** of vertices whose final shortest path weights from source s have already been determined, and a set **Q** of vertices whose shortest path weights are not yet known.

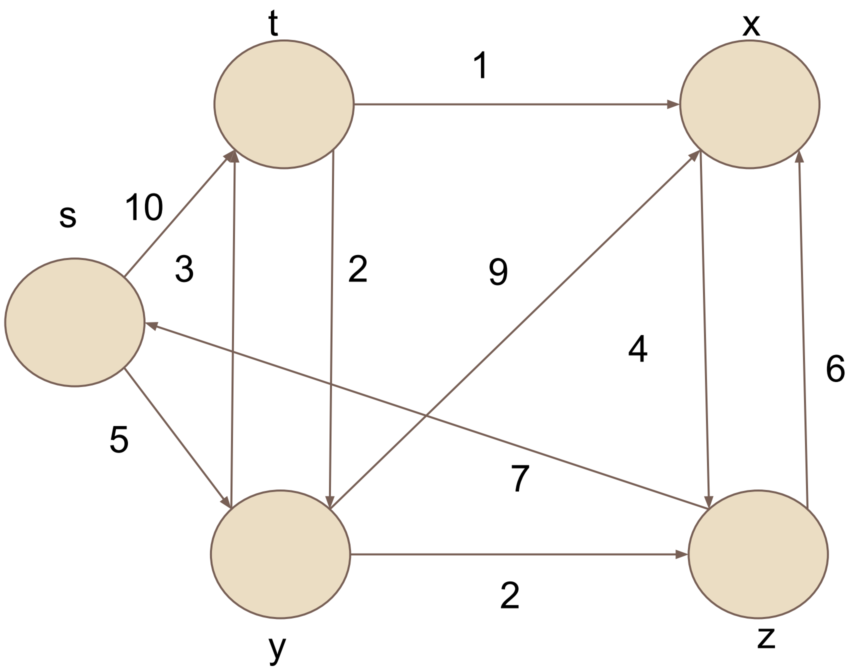
- Algorithm repeatedly selects the vertex v in **Q** with the minimum shortest path estimate.
  - Adds v to S.
  - Relaxes all the edges leaving v.

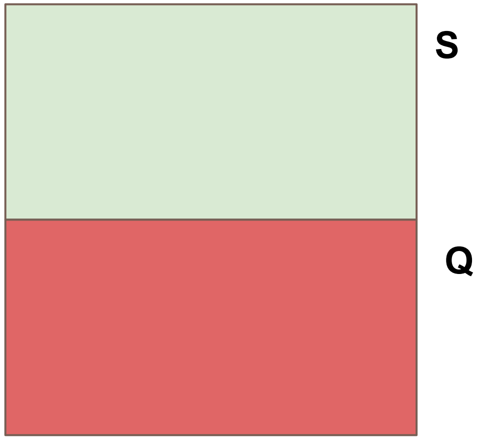
- We'll show in the proof that, at the point where we add v to S d(u,v) = **δ (u,v)**

Dijkstra's algorithm

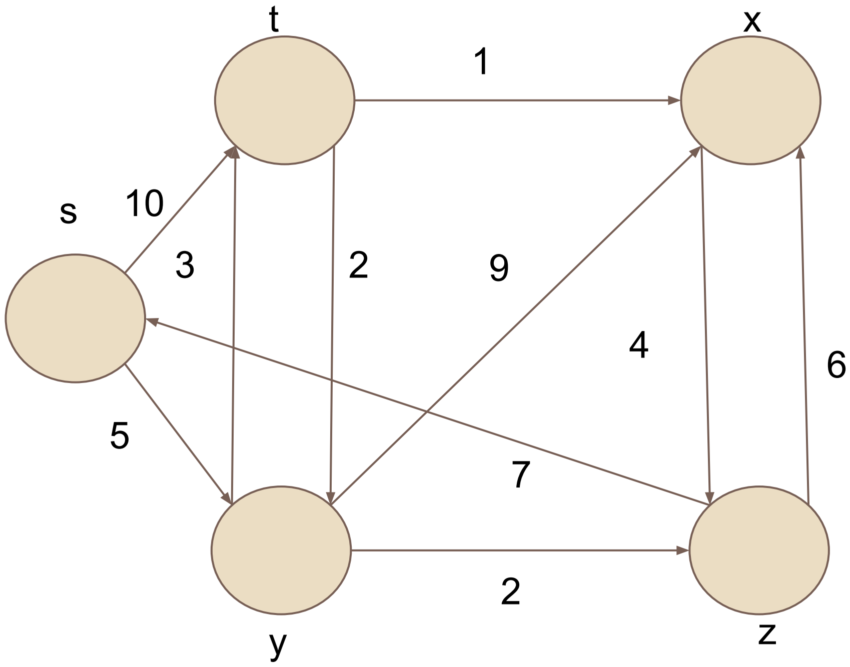


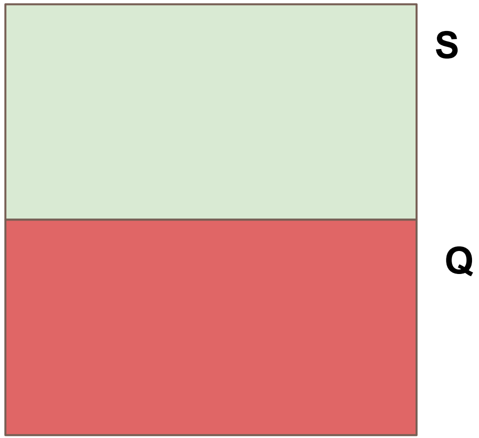
Dijkstra's algorithm





Dijkstra's algorithm

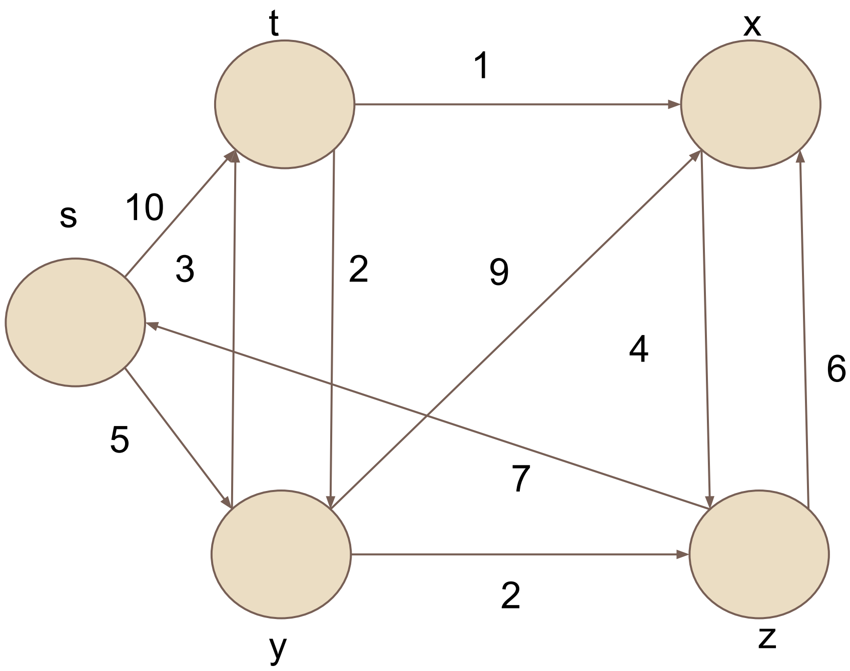


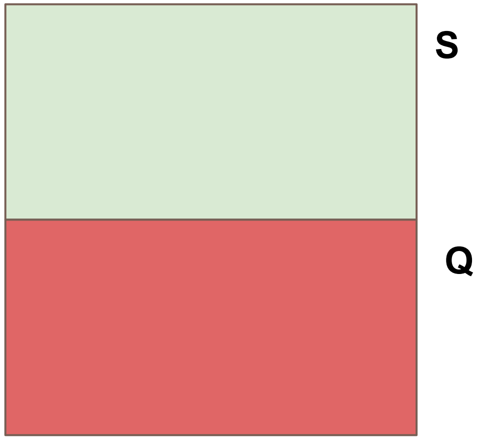


Initialisation

d[s,s] = ?
d[s,t] = ?
d[s,x] = ?

Dijkstra's algorithm





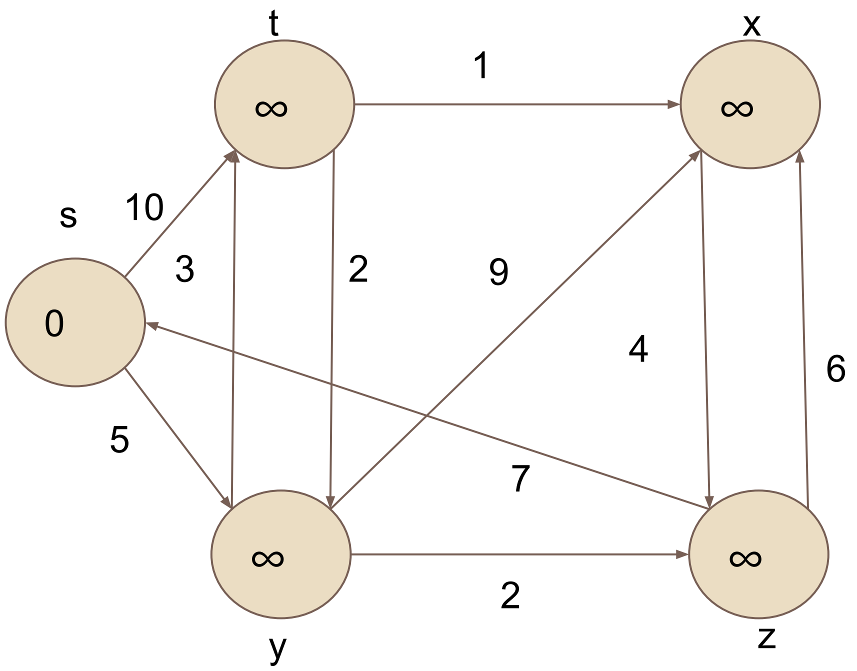
Initialisation

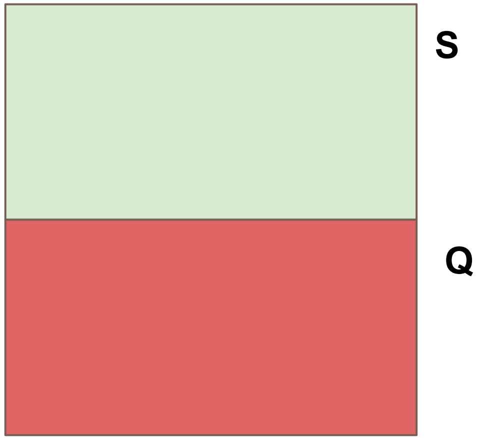
d[s,s] = 0

d[s,t] = ∞

d[s,x] = ∞

Dijkstra's algorithm





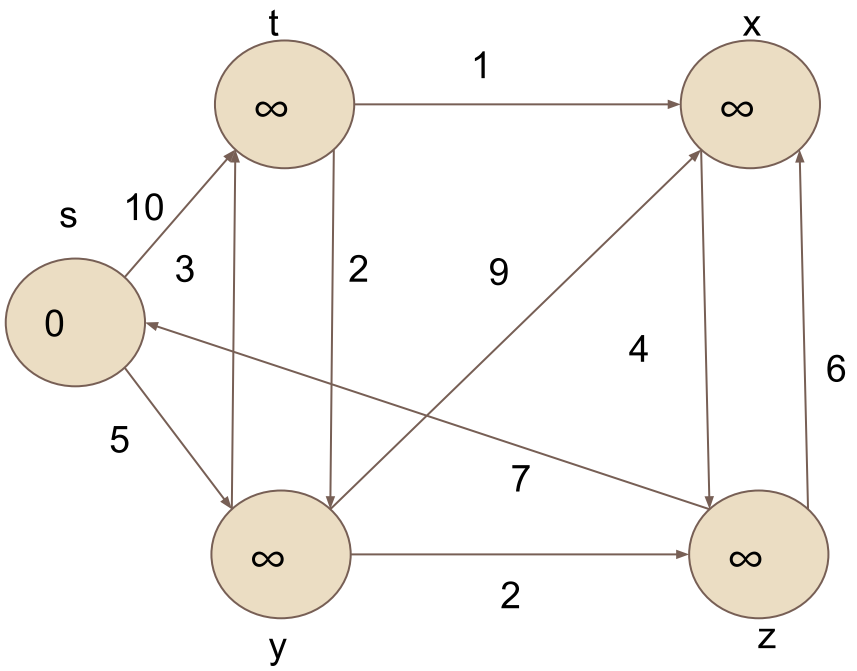
Initialisation

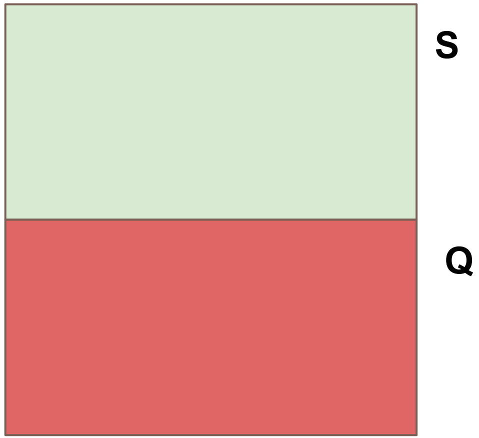
d[s,s] = 0

d[s,t] = ∞

d[s,x] = ∞

Dijkstra's algorithm





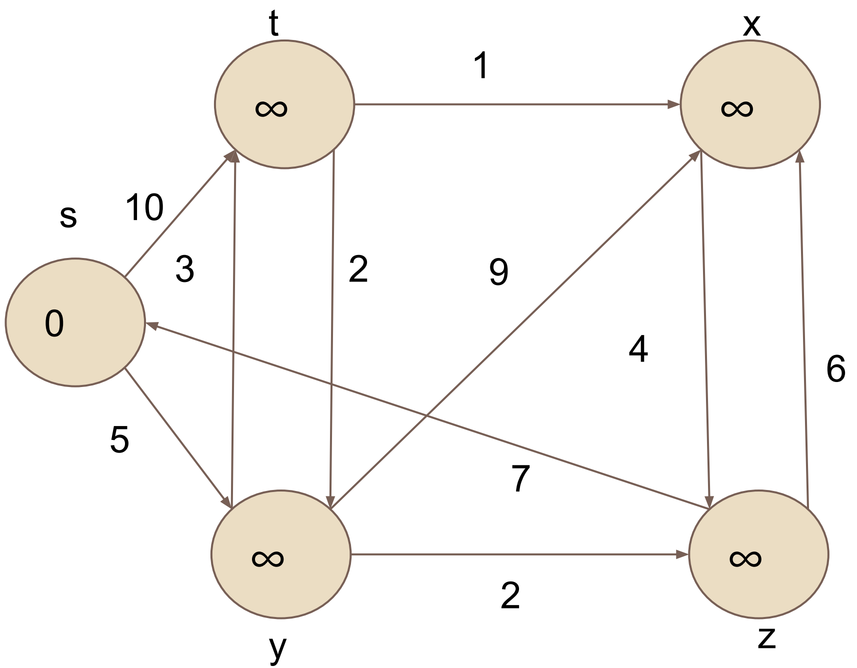
Initialisation

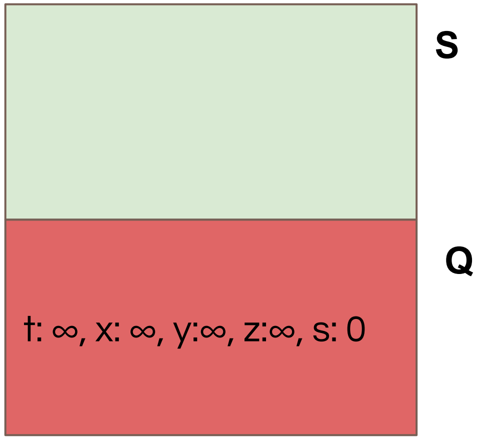
d[s,s] = 0 Π[s] = null

d[s,t] = ∞ Π[t] = null

d[s,x] = ∞ Π[x] = null

Dijkstra's algorithm

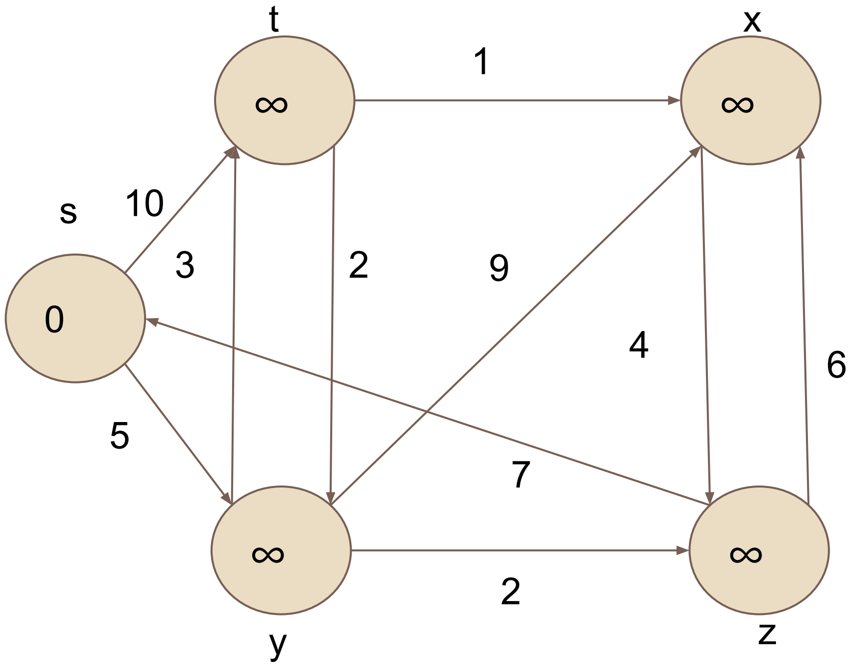


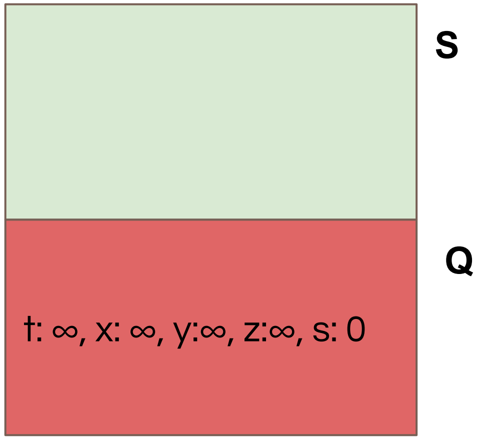


Initialisation

Place all node V in Q.

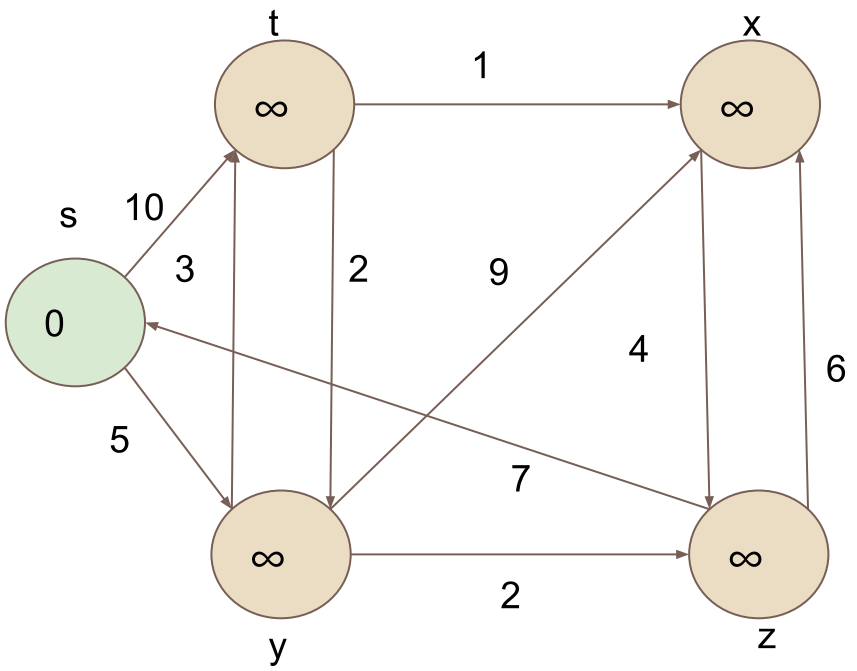
Dijkstra's algorithm





Pick node with smallest d[s,v] and place it in S

Dijkstra's algorithm



S

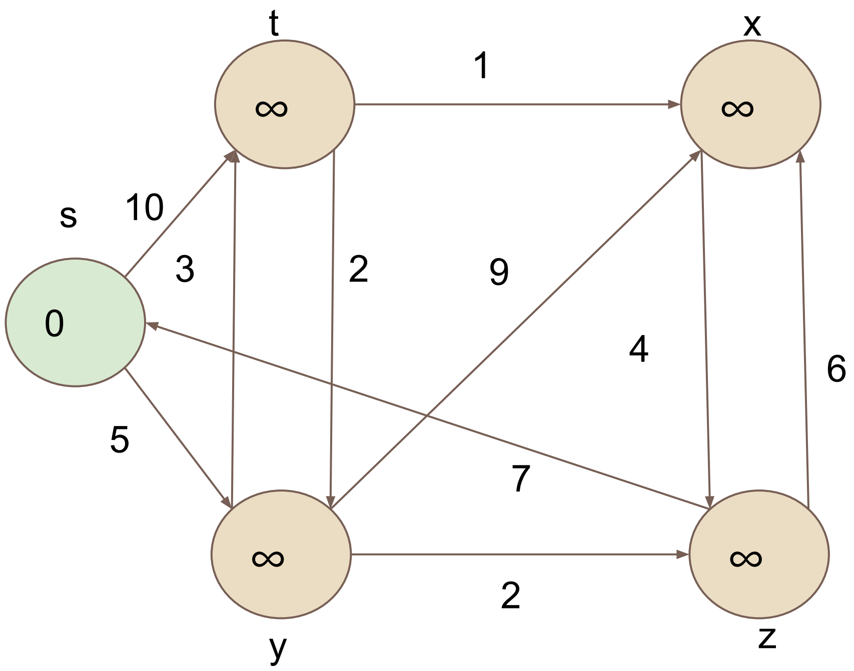
s: 0

Q

t: ∞, x: ∞, y:∞, z:∞

Pick node with smallest d[s,v] and place it in S

Dijkstra's algorithm



S

s: 0

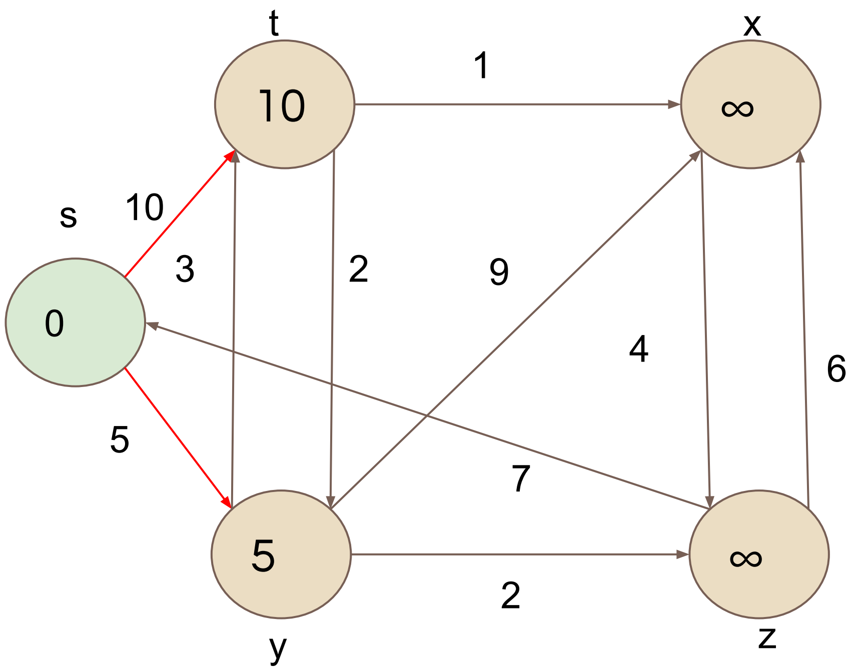
Q

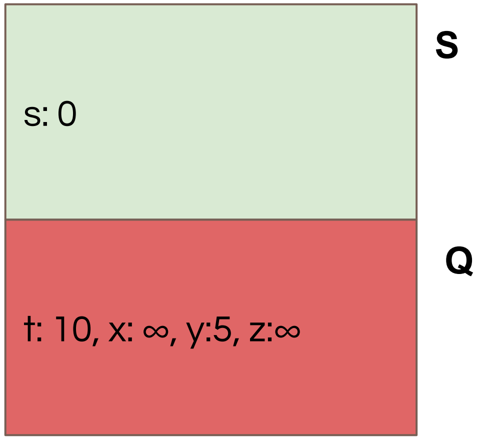
t: ∞, x: ∞, y:∞, z:∞

Pick node with smallest d[s,v] and place it in S

Relax all of its edges

Dijkstra's algorithm

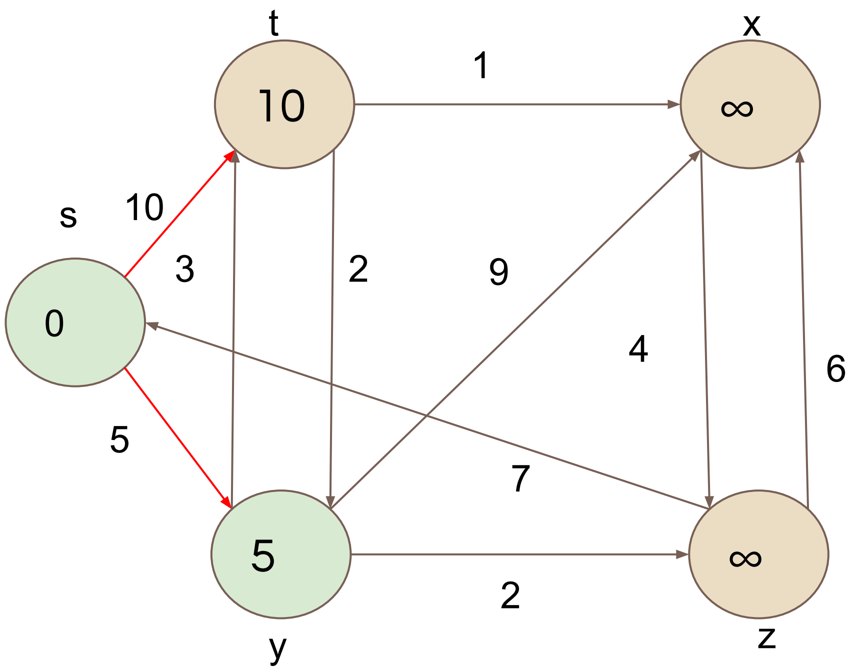




Pick node with smallest d[s,v] and place it in S

Relax all of its edges

Dijkstra's algorithm



S

s: 0, y:5

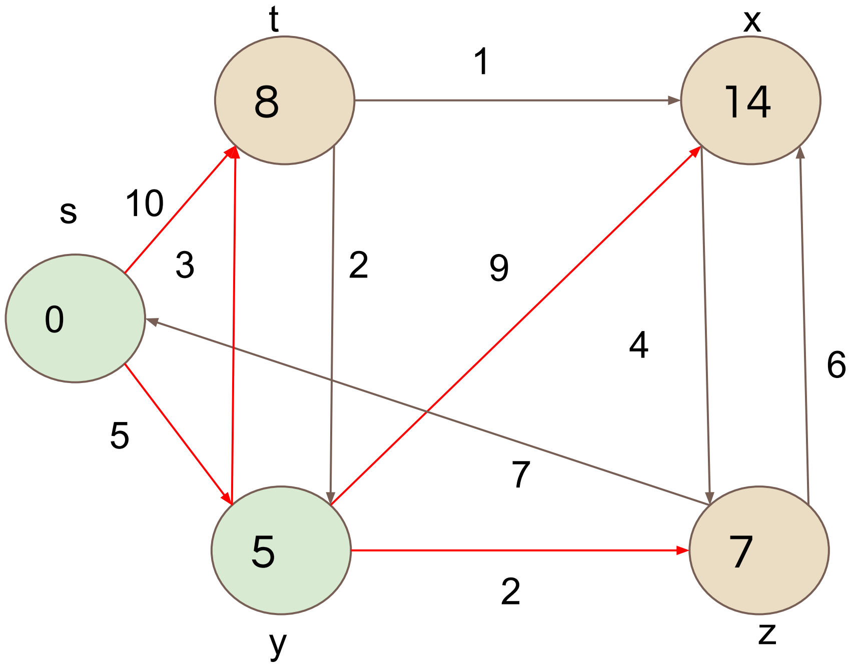
Q

t: 10, x: ∞, z:∞

Pick node with smallest d[s,v] and place it in S

Relax all of its edges

Dijkstra's algorithm



S

s: 0, y:5

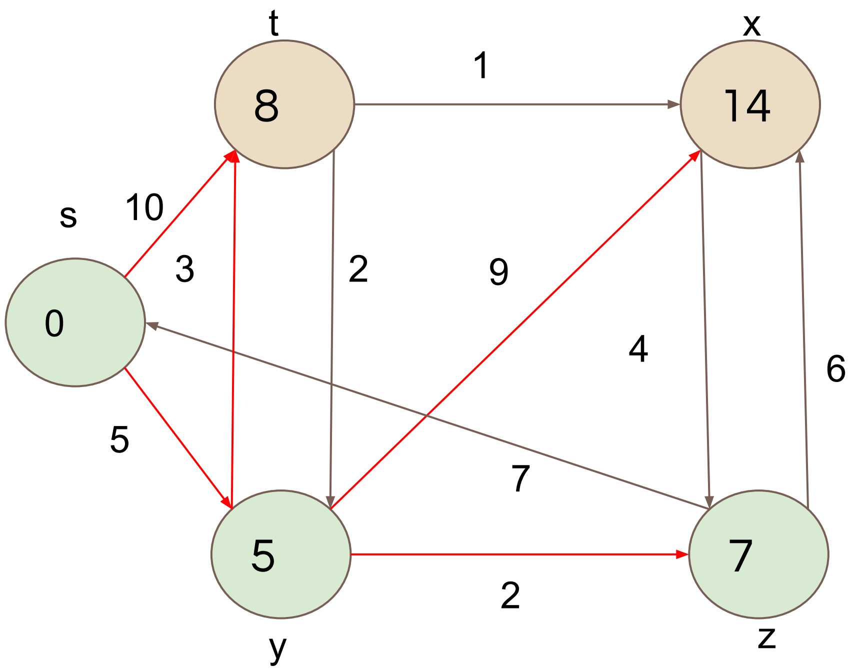
Q

t: 8, x: 14, z: 7

Pick node with smallest d[s,v] and place it in S

Relax all of its edges

Dijkstra's algorithm



S

s: 0, y:5, z: 7

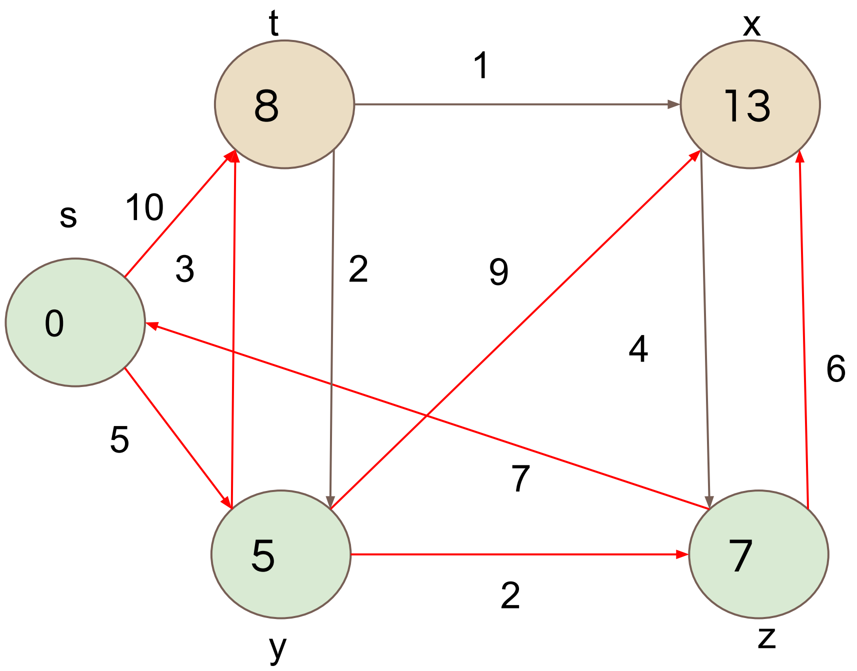
Q

t: 8, x: 14

Pick node with smallest d[s,v] and place it in S

Relax all of its edges

Dijkstra's algorithm



S

s: 0, y:5, z: 7

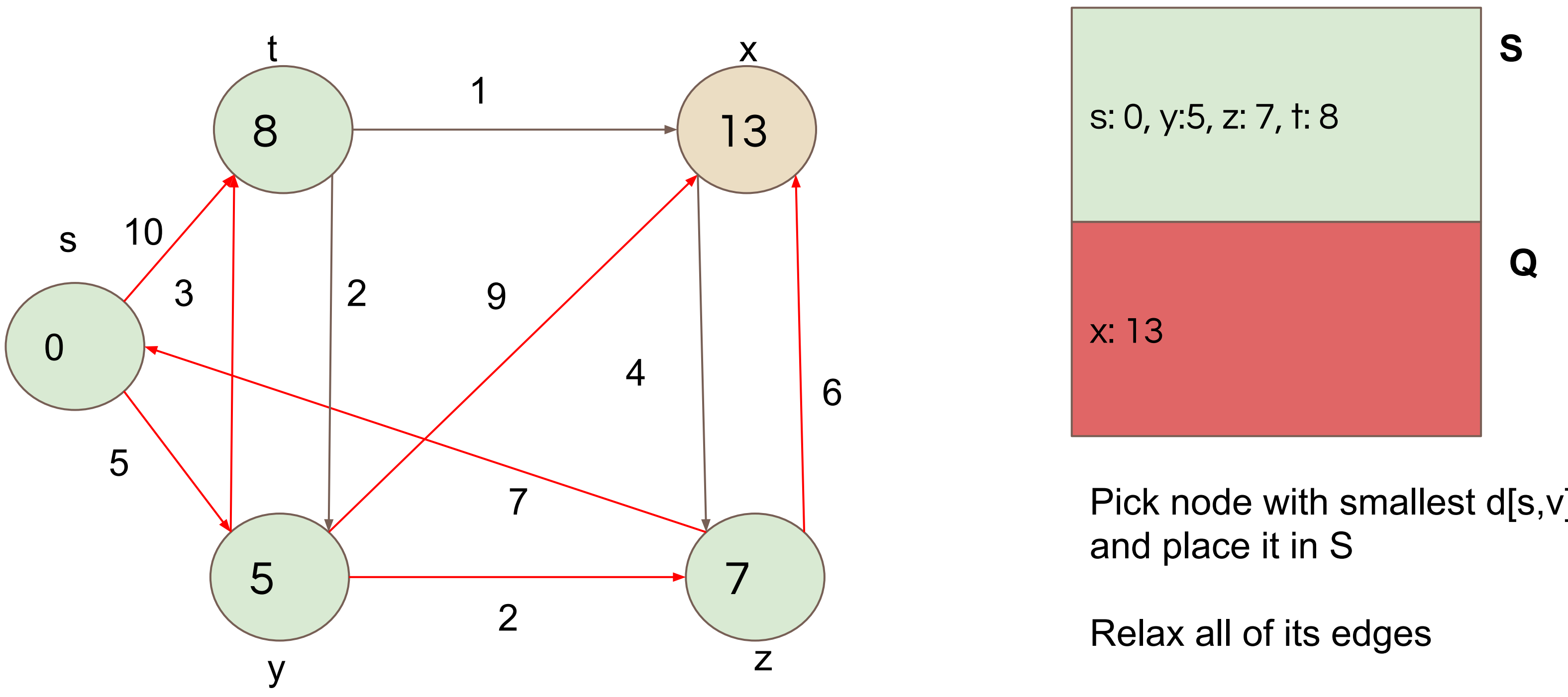
Q

t: 8, x: 13

Pick node with smallest d[s,v] and place it in S

Relax all of its edges

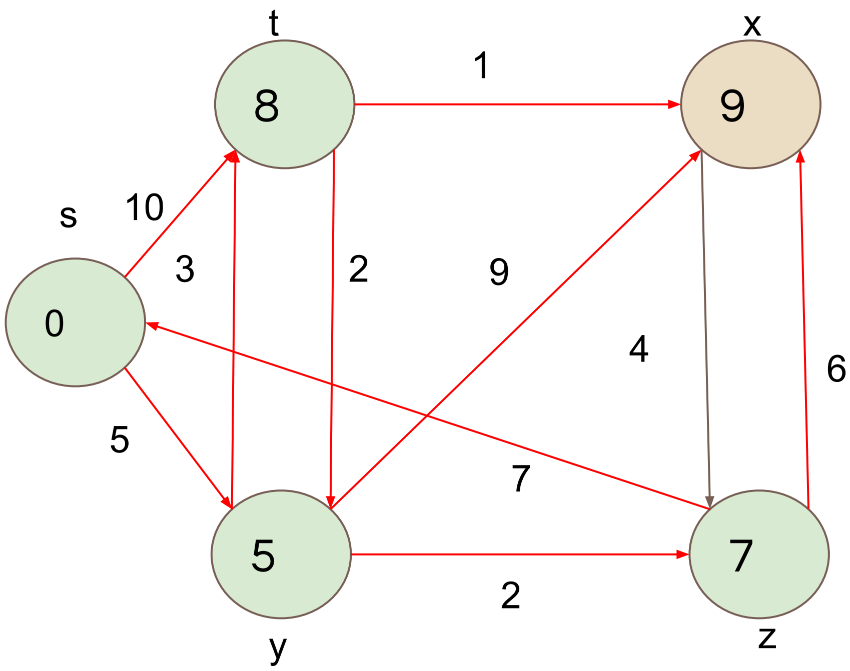
Dijkstra's algorithm



Pick node with smallest d[s,v] and place it in S

Relax all of its edges

Dijkstra's algorithm



S

s: 0, y:5, z: 7, t: 8

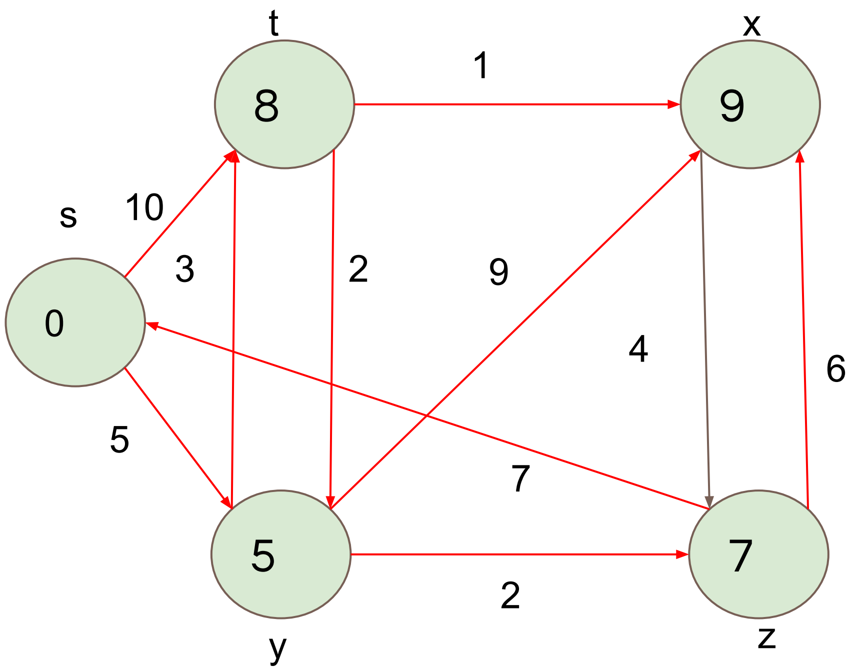
Q

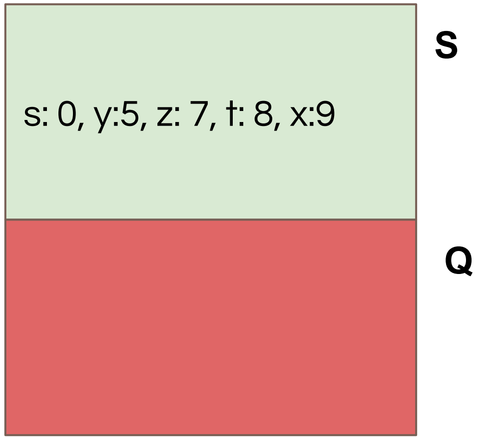
x: 9

Pick node with smallest d[s,v] and place it in S

Relax all of its edges

Dijkstra's algorithm

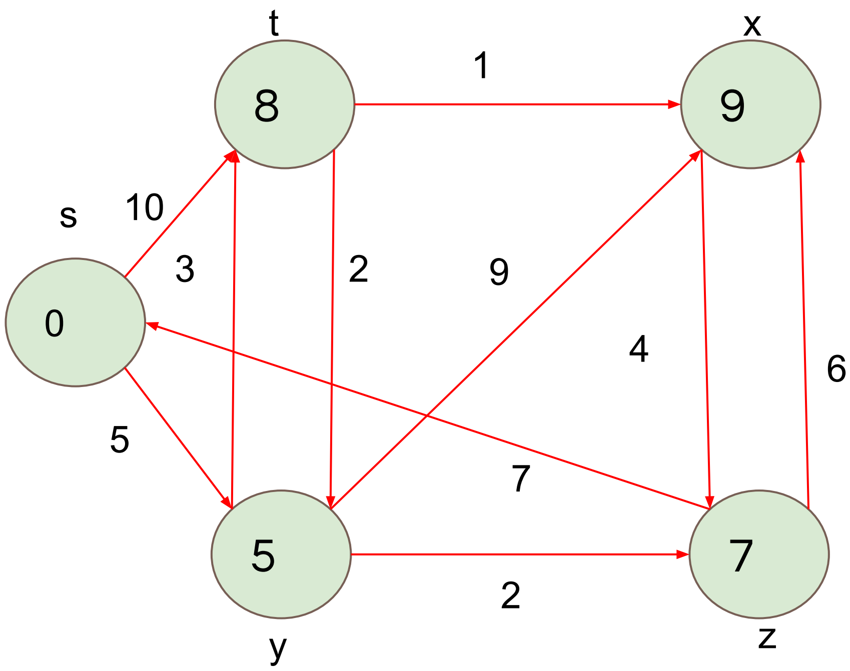


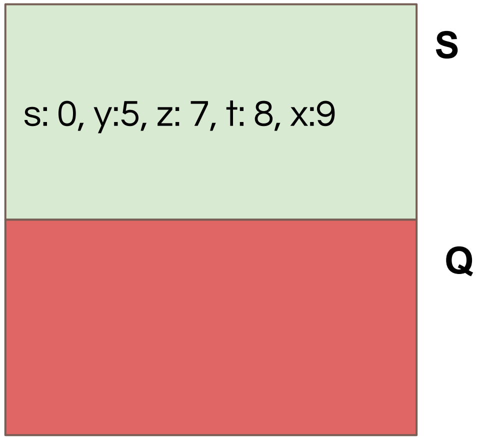


Pick node with smallest d[s,v] and place it in S

Relax all of its edges

Dijkstra's algorithm





Pick node with smallest d[s,v] and place it in S

Relax all of its edges

Dijkstra's algorithm

```
d[s,s] = 0
For v in V:
        d[s,v]= ∞
        Π[v] = null
S = ∅
Q = V
while Q ≠ ∅
        u = FindMinimum from Q
        S = S U {u}
        For each neighbour n of u:
                Relax(u,n)
```

```
Relax(u,n):
        If d[n] > d[u] + w(u,n):
                // Have discovered a shorter path
                d[n] = d[u] + w(u,n)
                // Update Predecessor of n
                Π[n] = u
                Update n in Q
        Else:
          // Already knew of a better path
```

Complexity

```
d[s,s] = 0
For v in V:
        d[s,v]= ∞
        Π[v] = null
S = ∅
Q = V
while Q ≠ ∅
        u = FindMinimum from Q
        S = S U {u}
        For each neighbour n of u:
                Relax(u,n)
```

```
Relax(u,n):
        If d[n] > d[u] + w(u,n):
                // Have discovered a shorter path
                d[n] = d[u] + w(u,n)
                // Update Predecessor of n
                Π[n] = u
                Update n in Q
        Else:
            // Already knew of a better path
```

Complexity

```
d[s,s] = 0
For v in V:
        d[s,v]= ∞
        Π[v] = null
S = ∅
Q = V
while Q ≠ ∅
        u = FindMinimum from Q
        S = S U {u}
        For each neighbour n of u:
                Relax(u,n)
```

Loop runs O(V) times

At most relax O(E) times

```
Relax(u,n):
        If d[n] > d[u] + w(u,n):
                // Have discovered a shorter path
                d[n] = d[u] + w(u,n)
                // Update Predecessor of n
                Π[n] = u
                Update n in Q
        Else:
            // Already knew of a better path
```

Complexity

```
d[s,s] = 0
For v in V:
        d[s,v]= ∞
        Π[v] = null
S = ∅
Q = V
while Q ≠ ∅
        u = FindMinimum from Q
        S = S U {u}
        For each neighbour n of u:
                Relax(u,n)
```

Call insert into Q O(V) times

Call FindMinimum O(V) times

Call Relax O(E) times.

```
Relax(u,n):
        If d[n] > d[u] + w(u,n):
                // Have discovered a shorter path
                d[n] = d[u] + w(u,n)
                // Update Predecessor of n
                Π[n] = u
                Update n in Q
        Else:
          // Already knew of a better path
```

Complexity - Priority Queue!

```
d[s,s] = 0
For v in V:
        d[s,v]= ∞
        Π[v] = null
S = ∅
Q = Insert(V,Q)
while Q ≠ ∅
        u = Extract-Min(Q)
        S = S U {u}
        For each neighbour n of u:
                Relax(u,n)
```

Call insert into Q O(V) times

Call Extract-Min O(V) times

Call DecreaseKey O(E) times.

```
Relax(u,n):
        If d[n] > d[u] + w(u,n):
                // Have discovered a shorter path
                d[n] = d[u] + w(u,n)
                // Update Predecessor of n
                Π[n] = u
                DecreaseKey(Q,n)
        Else:
            // Already knew of a better path
```

Complexity - Priority Queue!

```
d[s,s] = 0
For v in V:
        d[s,v]= ∞
        Π[v] = null
S = ∅
Q = Insert(V,Q)
while Q ≠ ∅
        u = Extract-Min(Q)
        S = S U {u}
        For each neighbour n of u:
                Relax(u,n)
```

Call insert into Q O(V) times

Call Extract-Min O(V) times

Call DecreaseKey O(E) times.

```
Relax(u,n):
        If d[n] > d[u] + w(u,n):
                // Have discovered a shorter path
                d[n] = d[u] + w(u,n)
                // Update Predecessor of n
                Π[n] = u
                DecreaseKey(Q,n)
        Else:
            // Already knew of a better path
```

Insert(v,Q): O(lg V)   Extract-Min: O(lg V)   Decrease-Key: O(lg V)

Complexity - Priority Queue!

```
d[s,s] = 0
For v in V:
        d[s,v]= ∞
        Π[v] = null
S = ∅
Q = Insert(V,Q)
while Q ≠ ∅
        u = Extract-Min(Q)
        S = S U {u}
        For each neighbour n of u:
                Relax(u,n)
```

Call insert into Q O(V) times

Call Extract-Min O(V) times

Call DecreaseKey O(E) times.

```
Relax(u,n):
        If d[n] > d[u] + w(u,n):
                // Have discovered a shorter path
                d[n] = d[u] + w(u,n)
                // Update Predecessor of n
                Π[n] = u
                DecreaseKey(Q,n)
        Else:
            // Already knew of a better path
```

O(V * lg V + V*lg V + E*lg(V))

Complexity - Priority Queue!

```
d[s,s] = 0
For v in V:
        d[s,v]= ∞
        Π[v] = null
S = ∅
Q = Insert(V,Q)
while Q ≠ ∅
        u = Extract-Min(Q)
        S = S U {u}
        For each neighbour n of u:
                Relax(u,n)
```

Call insert into Q O(V) times

Call Extract-Min O(V) times

Call DecreaseKey O(E) times.

```
Relax(u,n):
        If d[n] > d[u] + w(u,n):
                // Have discovered a shorter path
                d[n] = d[u] + w(u,n)
                // Update Predecessor of n
                Π[n] = u
                DecreaseKey(Q,n)
        Else:
            // Already knew of a better path
```

O(V * lg V + V*lg V + E*lg(V)) => O(V*lg V + V*lg V + E *O(1)) if use **Fibonacci Heaps**

Optimal Substructure

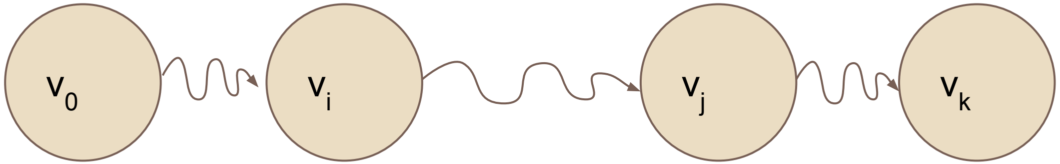
- Most shortest path algorithms rely on the **optimal substructure property**
  - Intuitively, says that **a shortest path between two vertices contains only other shortest paths within it**
  - If path p = (v_o, v_1,v_2) from v_0 to v_2 is the shortest path from v_0 to v_2, then (v_0,v_1) must also be the shortest path from v_o to v_1. Otherwise there'd be a better way to get to v_2!

Optimal Substructure

- Most shortest path algorithms rely on the **optimal substructure property**
  - Intuitively, says that **a shortest path between two vertices contains only other shortest paths within it**
  - If path p = (v_o, v_1,v_2) from v_0 to v_2 is the shortest path from v_0 to v_2, then (v_0, v_1) must also be the shortest path from v_o to v_1. Otherwise there'd be a better way to get to v_2!
- Given a graph G=(V,E,W), let p= (v_0,v_1, .. , v_k) be a shortest path from vertex v_o to vertex v_k and for any i and j such that 0<=i<=j<=k, let p_{ij} be the subpath of p from vertex v_i to vertex v_j. Then p_{ij} is the shortest path from v_i to v_j

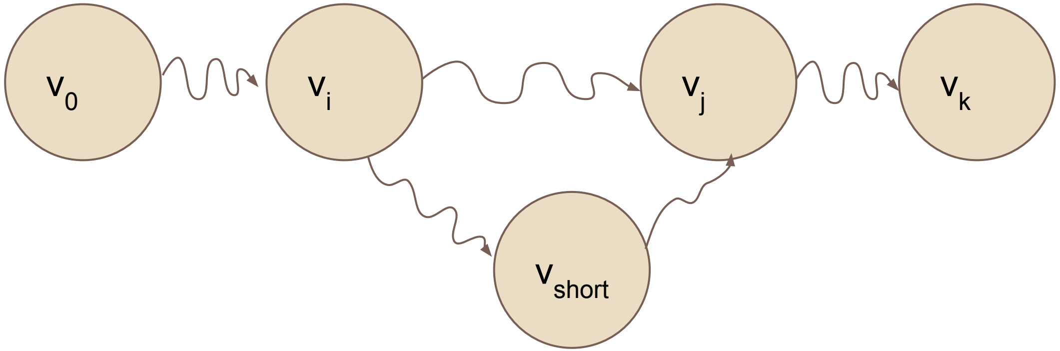
Optimal Substructure

- Proof by contradiction:
  - Assume that $p = (v_o, \ldots v_i .. v_j .. v_k)$ is the shortest path



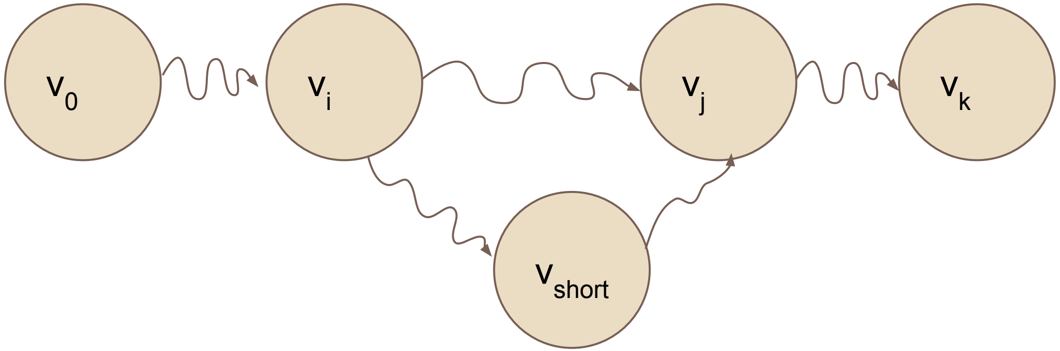
Optimal Substructure

- Proof by contradiction:
  - Assume that $p = (v_o, \ldots v_i .. v_j .. v_k)$ is the shortest path
  - Assume that there exists a shorter path between vertices i and vertices j.



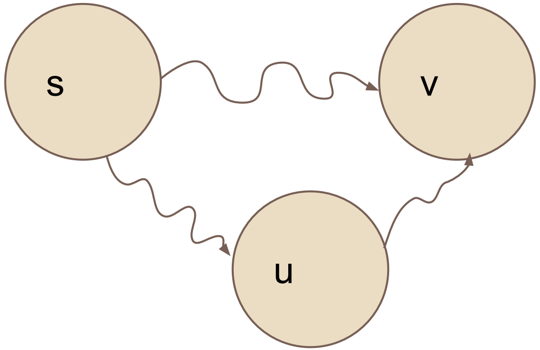
Optimal Substructure

- Proof by contradiction:
  - Assume that p = (v_o, … v_i .. v_j .. v_k) is the shortest path
  - Assume that there exists a shorter path between vertices i and vertices j.
  - Then the shortest path from v_0 to v_k would be via v_{short} so p is not the shortest path. **We have a contradiction**



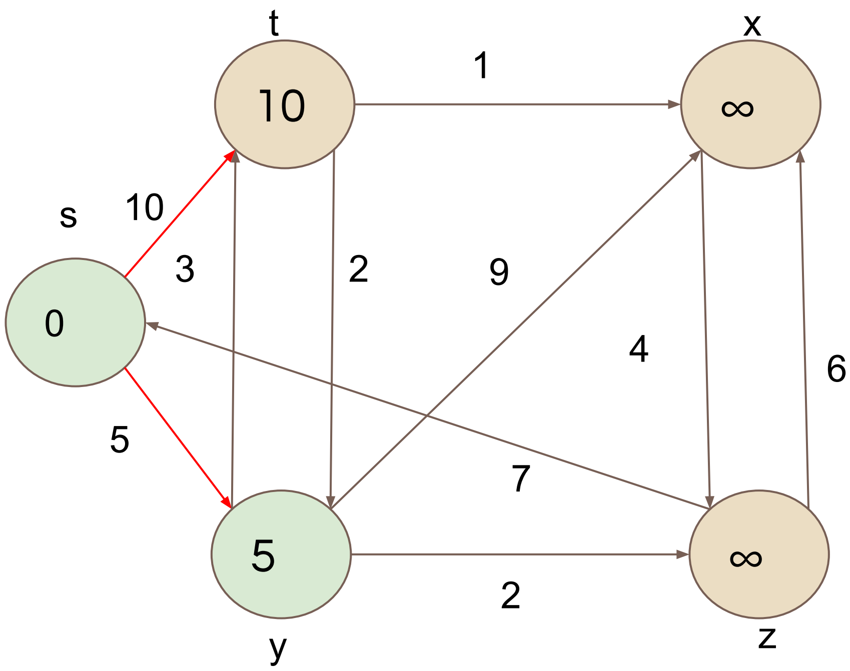
Triangle Inequality

- By the same logic, can derive the **triangle inequality**
  - **δ**(s,v) <= **δ**(s,u) + **δ**(u,v)



If the path (s .. v) is a shortest path, the weight of the path from (s,u) and from (u,v) cannot be smaller as that would mean that the path (s .. v) is not the shortest path

Dijkstra's algorithm - Again



Why is d[s,y] = **δ**(s,y)?

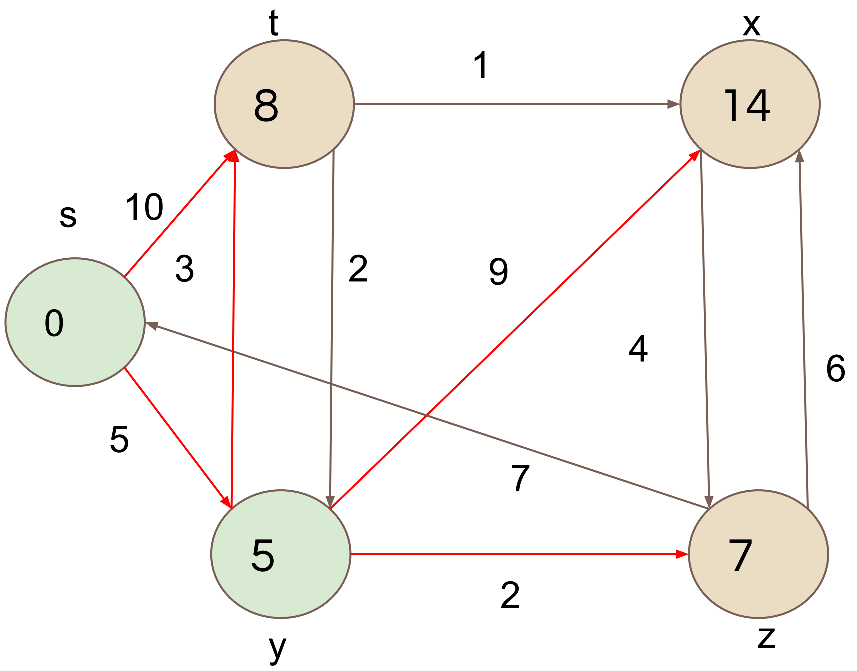
We have relaxed all the edges leaving s.

The only way to reach **y** is via (s,t) + (unknown path p) or via (s,y)

But w(s,t) > w(s,y) so w(s,t) + p > w(s,y) because w(p)>0

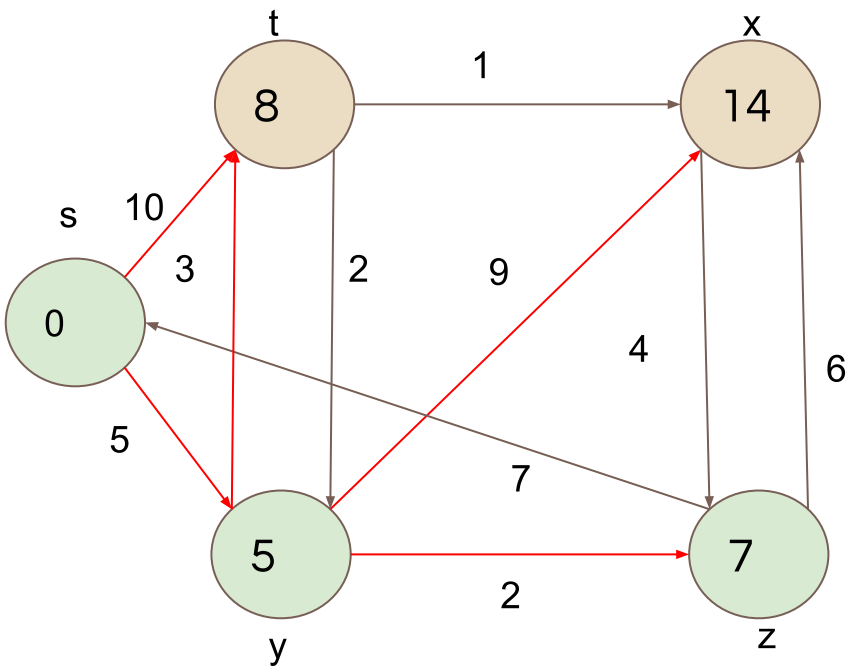
Any path that we take via t will have greater weight than w(s,y), so d[s,y] = **δ**(s,y)

Dijkstra's algorithm - Again



Now relax all of the edges that start from y, and update the current estimate of the shortest path.

Dijkstra's algorithm - Again



Why is d[s,z] = **δ**(s,z)?

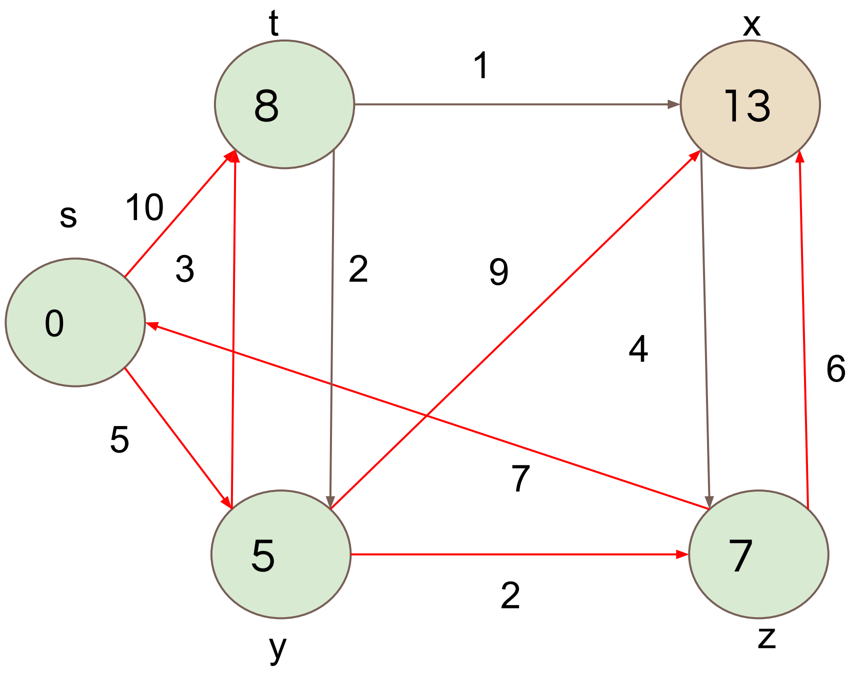
The current values represent **our best attempts to reach nodes t,x,z** using nodes s and y (because relaxed edges from s,y)

We want to show that reaching **z** through other nodes **t** and **x** would yield a value d that is greater than d[z].

Going through s,y,x (...) z would not lead a shorter path as d[s,x] = 14

Going through s,y,t (...) z (the current shortest path to t) would not lead a shorter path as d[s,t] = 8

Dijkstra's algorithm - Again



Why is d[s,t] = **δ**(s,t)?

The current values represent **our best attempts to reach nodes t,x** using nodes s,y,z (because relaxed edges from s,y,z)

We want to show that reaching **t** through other nodes **x** would yield a value d that is greater than d[t].

Going through s,y,z,x (the current shortest path to x) would not lead a shorter path as d[s,x] = 13

Correctness Proof (Intuition)

- Want to show that d[u,v] = $\boldsymbol{\delta}$(u,v)

Correctness Proof (Intuition)

- Want to show that d[u,v] = **δ**(u,v)

- **Lemma:** Initialising d[s] = 0 and d[v] = ∞ for all v ∈ V - {s} establishes d[v] ≥**δ**(s,v) for all v∈V, and this invariant is maintained over any sequence of relaxation steps. **Upper Bound Property**

Correctness Proof (Intuition)

- Want to show that d[u,v] = **δ**(u,v)

- **Lemma:** Initialising d[s] = 0 and d[v] = ∞ for all v ∈ V - {s} establishes d[v] ≥**δ**(s,v) for all v∈V, and this invariant is maintained over any sequence of relaxation steps. **Upper Bound Property**

- **Proof:**
  - At initialisation d[x] = ∞ so d[x] >= **δ**(u,x) for all x ∈ V
  - Assume, after i relaxation steps, that for all nodes x ∈ V, d[x] >= **δ**(u,x). And consider relaxing edge (x,v) (the (i+1)th relaxation step):
    - If we relax (x,v): d[v] = d[x] + w(x,v)
    - By assumption d[x]>= **δ**(u,x)
    - It follows that d[v] >= **δ**(u,x) + w(x,v).
    - It follows that d[v] >= **δ**(u,x) + **δ**(x,v). By definition, w(x,v)>=**δ**(x,v)
    - It follows that d[v] >= **δ**(u,x) + **δ**(x,v) >= **δ**(u,v) (by triangle inequality)

Correctness Proof (Intuition)

- **Theorem:** Dijkstra's algorithm terminates with d[v]=**δ**(s,v) for all in v∈V
- **Proof:** Want to show that d[v]=**δ**(s,v) for every v∈V when v is added to **S**

Correctness Proof (Intuition)

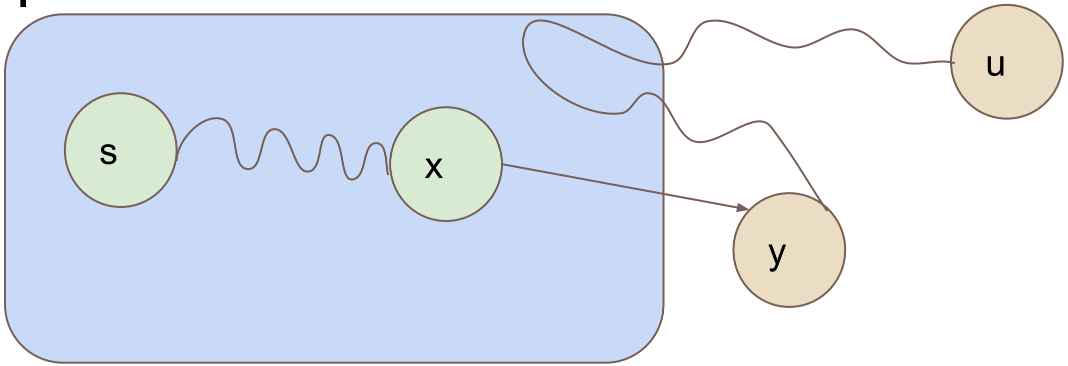
- **Theorem:** Dijkstra's algorithm terminates with d[v]=**δ**(s,v) for all in v∈V

- **Proof:** Want to show that d[v]=**δ**(s,v) for every v∈V when v is added to **S**
  - Suppose u is the first vertex added to **S** for which d[u] ≠ **δ**(s,u)
  - Let **y** be the first vertex in **Q** along a shortest path from **s** to **u**, and let **x** be its **predecessor**

Correctness Proof (Intuition)

- **Theorem:** Dijkstra's algorithm terminates with d[v]=**δ**(s,v) for all in v∈V
- **Proof:** Want to show that d[v]=**δ**(s,v) for every v∈V when v is added to **S**
  - Suppose u is the first vertex added to **S** for which d[u] ≠ **δ**(s,u)
  - Let **y** be the first vertex in **Q** along a shortest path from **s** to **u**, and let **x** be its **predecessor**

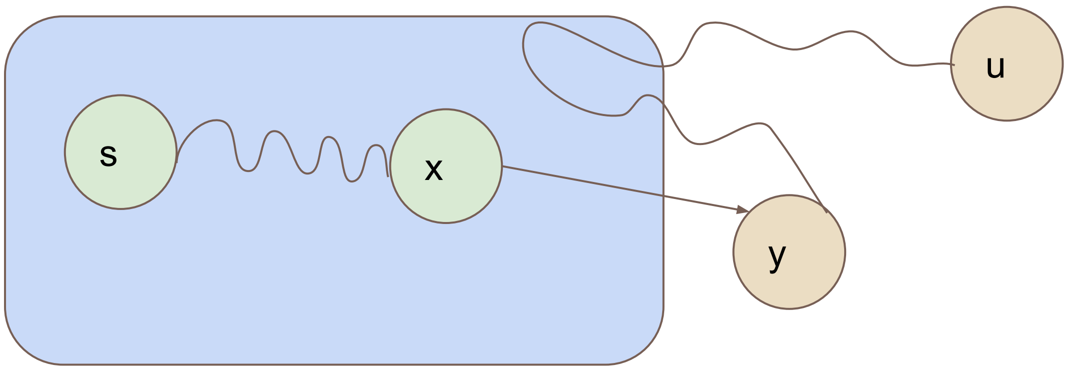
S, just before adding **u**



Correctness Proof (Intuition)

- Since u is the first vertex violating the invariant, we have d[x] = **δ**(s,x)
- Since subpaths of shortest paths are shortest paths, and y is on shortest path from s to u, d[y] was set to **δ**(s,x) + w(x,y) = **δ**(s,y) just after x was added to s
- We have d[y] = **δ**(s,y) and **δ**(s,y) <= **δ**(s,u) <= d[u] (Upper Bound Property)

S, just before adding **u**



Correctness Proof (Intuition)

- But, d[y] ≥ d[u] since the algorithm chose u first
- Hence d[y] = **δ**(s,y) = **δ**(s,u) = d[u]
- We have a contradiction! So d[u] = **δ**(s,u)

S, just before adding **u**