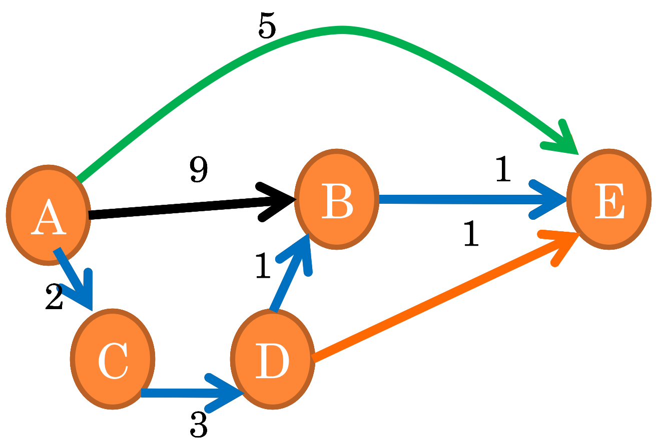
Shortest Path

Tanjina Helaly

CSI 207: Algorithms

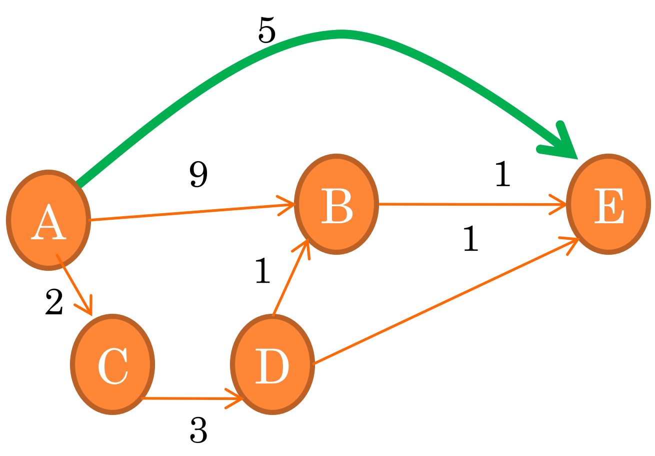
Shortest Path

- Applicable for directed weighted graph.
- Available paths from A to E
  - A->B->E = 10
  - A->C->D->E=6
  - A->C->D->B->E=6
  - A->E = 5



Shortest Path

- Applicable for directed weighted graph.
- Available paths from A to E
  - A->B->E = 10
  - A->C->D->E=6
  - A->C->D->B->E=6
  - A->E = 5
- Shortest among all available path = 5



Shortest Path - definition

- ***Given** a weighted, directed graph* G = (V,E) , with weight function w : E -> R mapping edges to real-valued weights.
- The ***weight w(p) of path p = <v_0, v_1, . . ., v_k> is** the sum of the weights* of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \quad where\ (v_{i-1}, v_i) \in E$$

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

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v , \\ \infty & \text{otherwise .} \end{cases}$$

- A *shortest path from vertex u to vertex v is then defined as any path p with weight*

  w(p) =δ(u,v)

Shortest Path - variants

- 3 variants
  - Source-Sink: From one vertex to another
  - Single source/Destination: from one vertex to every other
    - Example : GPS. Will find
  - All pairs: between all pairs of vertices
- Restrictions on edge weight
  - Non-negative
  - Arbitrary weight
  - Euclidean weight
- Cycles?
  - No directed cycle
  - No negative cycle.

Assumption

- There is path from s to every other vertices.

Applications

- Google map to find shortest distance from one location to another
- Robot navigation
- Urban traffic planning
- Optimal pipelining of VLSI chip

How to solve

- There are exponential number of ways to get from one vertex to another.

- So, try all possible combination and select the shortest one.

Generic Algorithm

- Algorithm:

  Initialize d[s]=0 and d[v]=∞ for all other vertices.

  Repeat until optimal condition is satisfied, select edge (u,v)

  Relax edge (u,v)

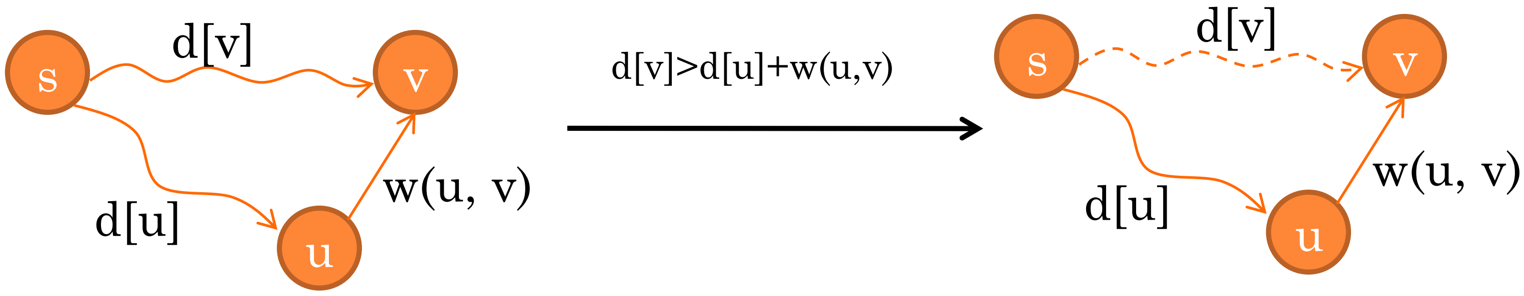
Relaxation

- Relaxation:

  If d[v]>d[u]+w(u,v)

  d[v]= d[u]+w(u,v)

  ∏[v]=u



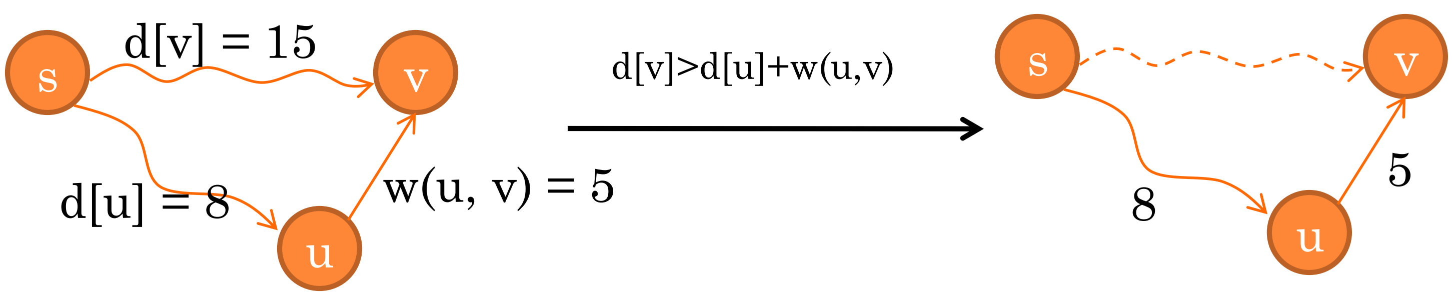
Relaxation – example

- Relaxation:

  If d[v]>d[u]+w(u,v)

  d[v]= d[u]+w(u,v)

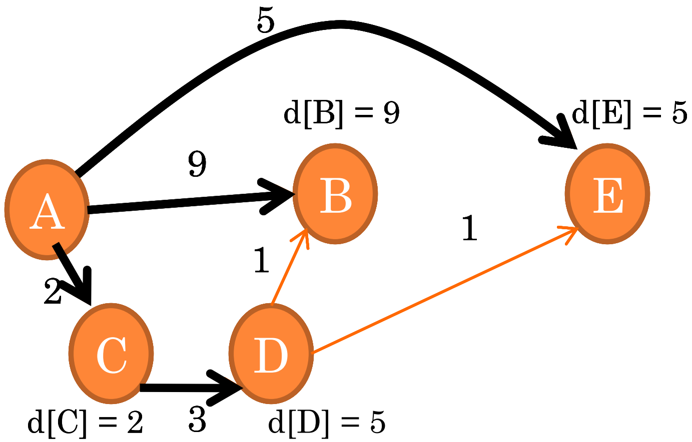
  ∏[v]=u



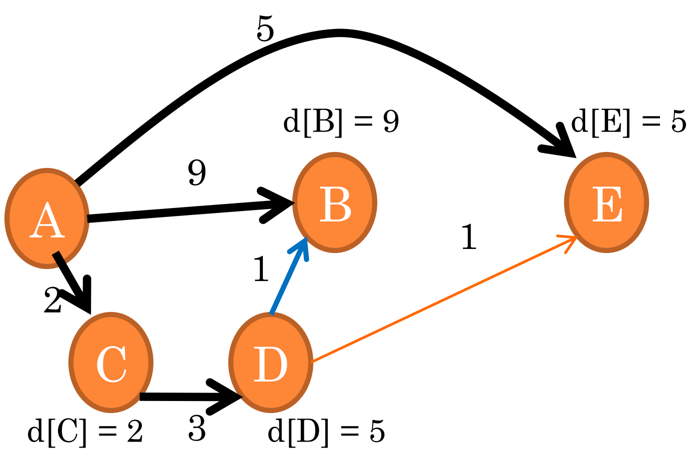
Optimal Condition

- For all edges, $d[v] \leq d[u] + w(u, v)$ [i.e. we found shortest path for all vertices]

Generic Algorithm



Generic Algorithm



d[D]+w(D,B) =
5+1=6 < d[B]

So there is a shorter way to get to B via D node.

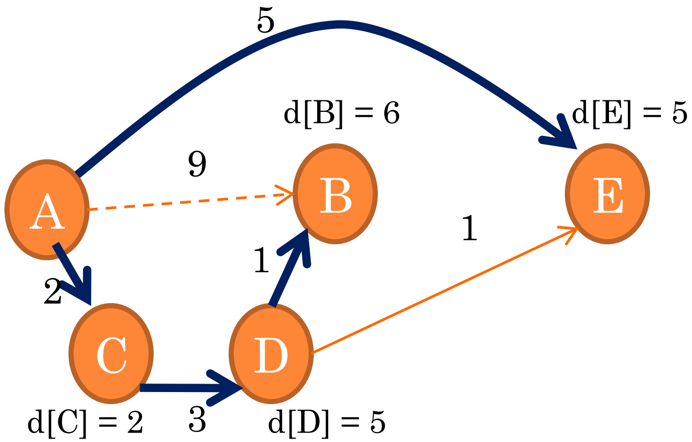
Generic Algorithm

- Relaxation:

  If d[v]>d[u]+w(u,v)

  d[v]= d[u]+w(u,v)

  ∏[v]=u



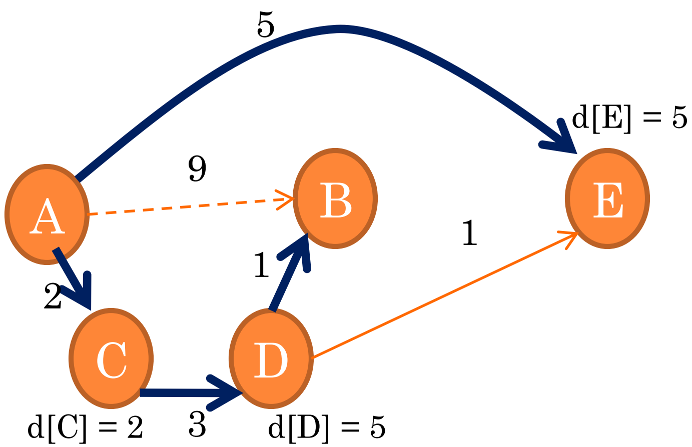
d[D]+w(D,B) =
5+1=6 < d[B]

So, Relax edge (D,B)
-means remove the
AB path and add DB
to get to node B

Generic Algorithm

- Optimal Condition:

  For all edges, d[v]≤d[u] +w(u, v) [i.e. we found shortest path for all vertices]

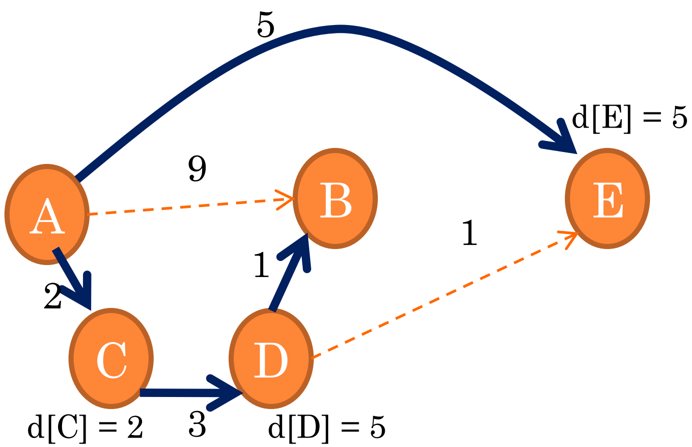


Haven't check the (D,E) edge yet.

Generic Algorithm

- Optimal Condition:

  For all edges, $d[v] \leq d[u] + w(u, v)$ [i.e. we found shortest path for all vertices]



$d[E] < d[D] + w(D,E)$.
So, no need to relax.

Generic Algorithm

- Efficient Implementations:
  - How to choose which edge to relax?
  - Ans: Greedy Algorithm
    - Dijkstra's algorithm (non-negative weights)
      - $O(V \lg V+E) = O(E)$ as $E=O(V^2)$
    - Bellman-Ford algorithm (no negative cycles)
      - $O(VE)$

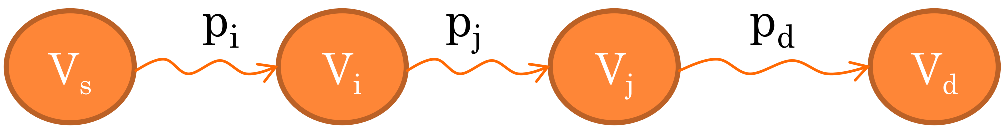
Some notation

- d(v) – distance to v from source node
- ∏[v]: predecessor on current best path to v.
- δ(v) – shortest path from source to v.

Greedy Algorithm

Optimal substructure

- Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.
- Subpath of a shortest path is also a shortest path
- If shortest path from s to d,
  - $\delta(s, d) = p_i + p_j + p_d$
  - Then p_i , p_j , and p_d are also shortest path

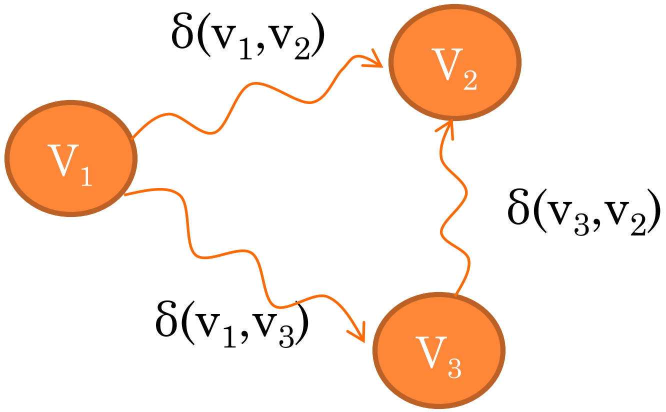


OPTIMAL SUBSTRUCTURE

- Formal def:
  - Given a weighted, directed graph G = (V,E) with weight function w : E -> R, let $\boldsymbol{p = (v_0, v_1, \ldots, v_k)}$ be a shortest path from vertex 0 to vertex k and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} = \boldsymbol{(v_i, v_{i+1}, \ldots, v_j)}$ be the subpath of p from vertex i to vertex j . Then, p_{ij} is a shortest path from i to j .

Triangle inequality

- If v_1, v_2, v_3 forms a triangle and $\delta(v_1,v_2)$, $\delta(v_1,v_3)$ and $\delta(v_3,v_2)$ are shortest paths between 2 vertices than we can write
  - $\delta(v_1,v_2) < \delta(v_1,v_3) + \delta(v_3,v_1)$



Dijkstra's Algorithm

Dijkstra's Algorithm

- Requirement/criteria
  - all edge weights are nonnegative.
- It maintain 2 lists.
  - set S of vertices whose final shortest-path weights from the source s have already been determined.
  - Another list Q for the vertices shortest-path weights has not been determined.
  - At each pass a vertex u will move from Q to S.
    - And relaxes all edges leaving u

DIJKSTRA'S ALGORITHM

$\text{DIJKSTRA}(G, w, s)$

$\text{INITIALIZE-SINGLE-SOURCE}$

$S = \phi;$

$\text{for each vertex } v \in G.V \;\; d[v] = \infty;$

$d[s] = 0$

$Q = G.V \quad$ [priority queue where key is d[v]]

$\text{while } Q \neq \phi$

$\quad u = \text{EXTRACT-MIN}(Q)$ ← Greedy Choice

$\quad S = S \bigcup \{u\}$

$\quad \text{for each vertex } v \in G.\text{Adj}(u)$

$\quad\quad \text{RELAX}(u, v, w)$

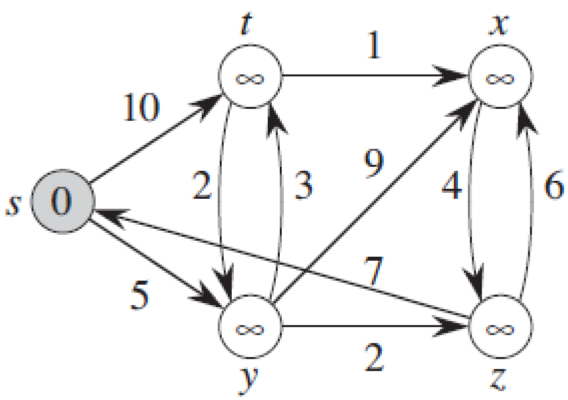
$\text{RELAX}(u, v, w):$

$\quad \text{if } (d[v] > d[u] + w(u, v))$

$\quad\quad d[v] = d[u] + w(u, v)$

$\quad\quad \pi[v] = u$

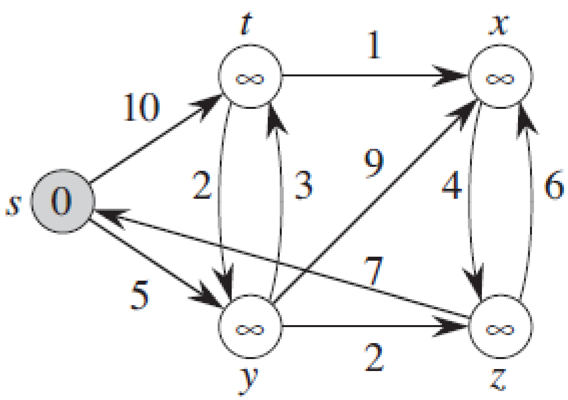
Dijkstra's Algorithm - example



S=S U {u}	Q {s t x y z}	{u}
{Φ}	{0 ∞ ∞ ∞ ∞}	

|Q|=5

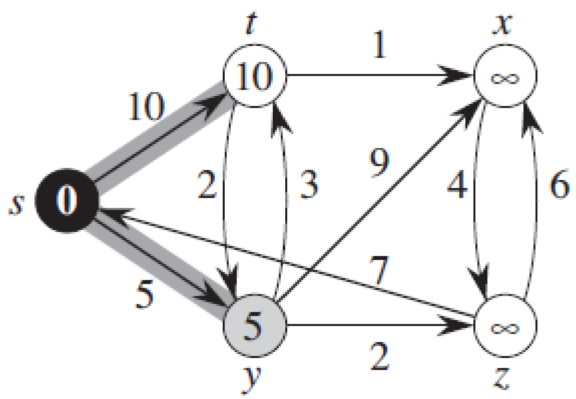
Dijkstra's Algorithm - example



S=S U {u}	Q {s t x y z}	{u}
{Φ}	{**0** ∞ ∞ ∞ ∞}	{s}

|Q|=5

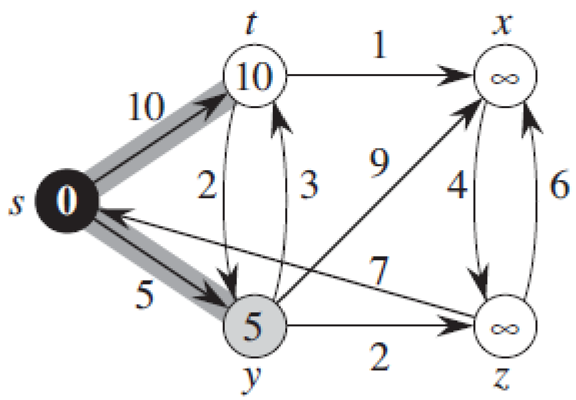
Dijkstra's Algorithm - example



S=S U {u}	Q {s t x y z}	{u}
{Φ}	{**0** ∞ ∞ ∞ ∞}	{s}
{s}	{**0** 10 ∞ 5 ∞}	

\|Q\|=4

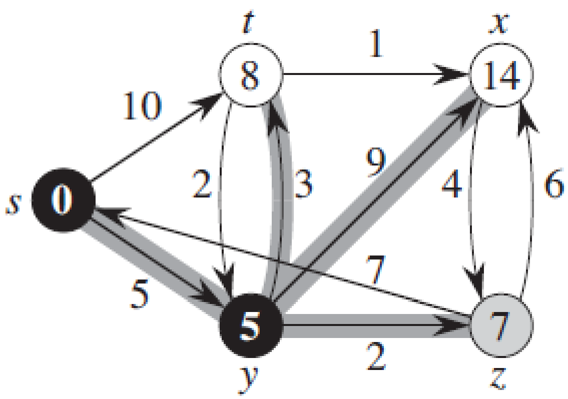
Dijkstra's Algorithm - example



S=S U {u}	Q {s t x y z}	{u}
{Φ}	{**0** ∞ ∞ ∞ ∞}	{s}
{s}	{**0** 10 ∞ **5** ∞}	{y}

|Q|=4

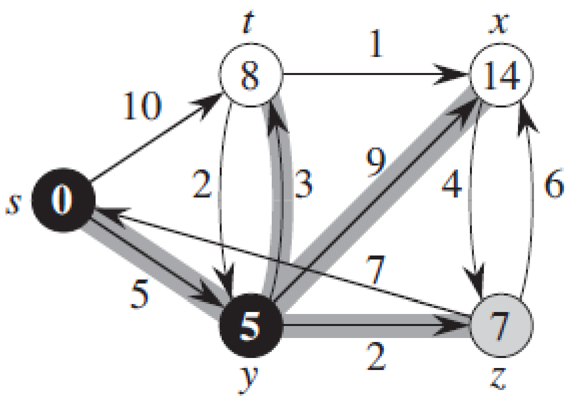
Dijkstra's Algorithm - example



S=S U {u}	Q {s t x y z}	{u}
{Φ}	{**0** ∞ ∞ ∞ ∞}	{s}
{s}	{**0** 10 ∞ **5** ∞}	{y}
{s, y}	{**0** 8 14 **5** **7**}	

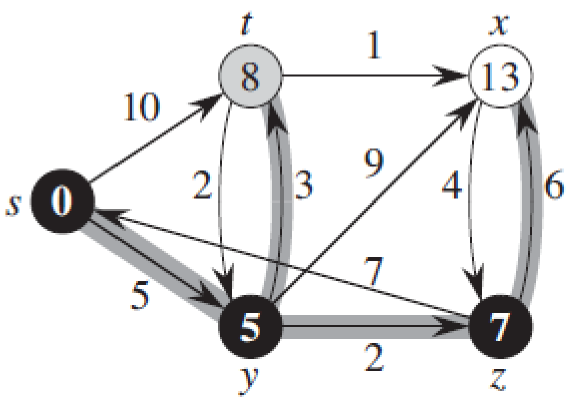
|Q|=3

Dijkstra's Algorithm - example



S=S U {u}	Q {s t x y z}	{u}	
{Φ}	{**0** ∞ ∞ ∞ ∞}	{s}	
{s}	{**0** 10 ∞ **5** ∞}	{y}	
{s, y}	{**0** 8 14 **5** **7**}	{z}	|Q|=3

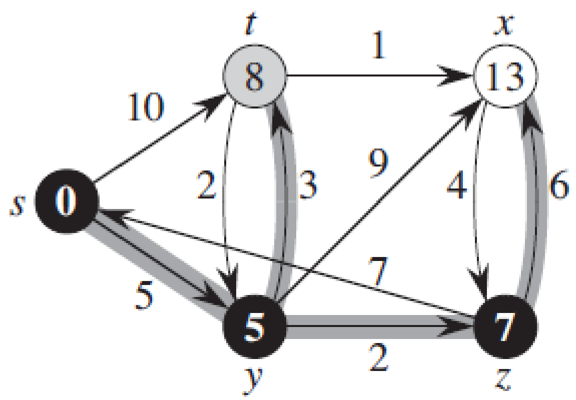
Dijkstra's Algorithm - example



S=S U {u}	Q {s t x y z}	{u}
{Φ}	{0 ∞ ∞ ∞ ∞}	{s}
{s}	{0 10 ∞ 5 ∞}	{y}
{s, y}	{0 8 14 5 7}	{z}
{s, y, z}	{0 8 13 5 7}	

\|Q\|=2

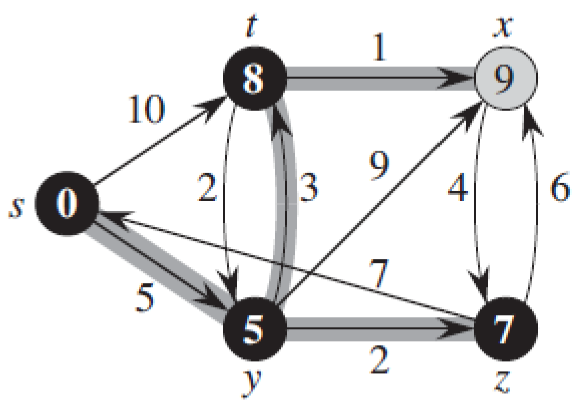
Dijkstra's Algorithm - example



S=S U {u}	Q {s t x y z}	{u}
{Φ}	{**0** ∞ ∞ ∞ ∞}	{s}
{s}	{**0** 10 ∞ **5** ∞}	{y}
{s, y}	{**0** 8 14 **5** **7**}	{z}
{s, y, z}	{**0** **8** 13 **5** **7**}	{t}

|Q|=2

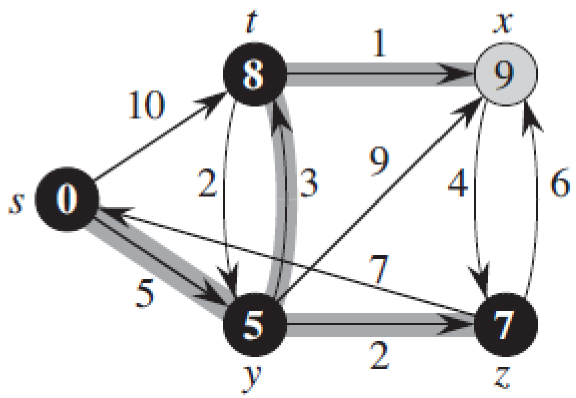
Dijkstra's Algorithm - example



S=S U {u}	Q {s t x y z}	{u}
{Φ}	{**0** ∞ ∞ ∞ ∞}	{s}
{s}	{**0** 10 ∞ **5** ∞}	{y}
{s, y}	{**0** 8 14 **5** **7**}	{z}
{s, y, z}	{**0** **8** 13 **5** **7**}	{t}
{s, y, z, t}	{**0** **8** 9 **5** **7**}	

|Q|=1

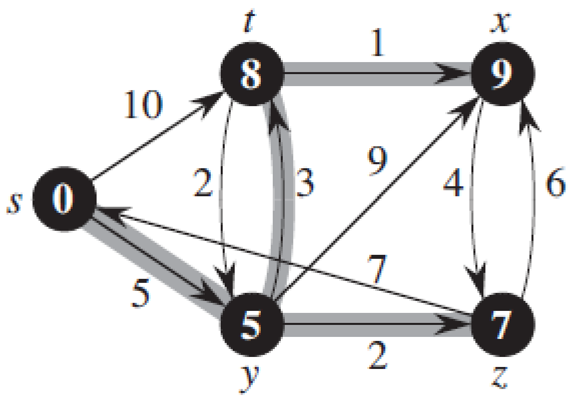
Dijkstra's Algorithm - example



S=S U {u}	Q {s t x y z}	{u}
{Φ}	{0 ∞ ∞ ∞ ∞}	{s}
{s}	{0 10 ∞ 5 ∞}	{y}
{s, y}	{0 8 14 5 7}	{z}
{s, y, z}	{0 8 13 5 7}	{t}
{s, y, z, t}	{0 8 9 5 7}	{x}

\|Q\|=1

DIJKSTRA'S ALGORITHM - EXAMPLE



S=S U {u}	Q {s t x y z}	{u}
{Φ}	{**0** ∞ ∞ ∞ ∞}	{s}
{s}	{**0** 10 ∞ **5** ∞}	{y}
{s, y}	{**0** 8 14 **5** **7**}	{z}
{s, y, z}	{**0** **8** 13 **5** **7**}	{t}
{s, y, z, t}	{**0** **8** **9** **5** **7**}	{x}
{s, y, z, t,x}	{**0** **8** **9** **5** **7**}	

|Q|=0

Time complexity

- Depends on how many times each operation takes place:
  - Inserts in Priority queue – V times
  - Extract min operation – V times
  - Decrease key operation(relaxation) - E times

Time complexity – with different data structure

- With Array
  - $\Theta(1)$ – insert
  - $\Theta(V)$- Extract min
  - $\Theta(1)$ – decrease key
  - Total = $V^* \Theta(1) + V^*\Theta(V) + E^*\Theta(1) = \Theta(E) + \Theta(V^2) = \Theta(V^2)$

- Binary mean heap
  - $\Theta(\lg V)$ – insert
  - $\Theta(\lg V)$- Extract min [deleting min is constant but resorting/updating after delete will take lg V]
  - $\Theta(\lg V)$ – decrease key
  - Total = $V^* \Theta(\lg V) + V^*\Theta(\lg V) + E^*\Theta(\lg V) = \Theta(E \lg V)$

- Fibonacci heap :
  - $\Theta(1)$ – insert
  - $\Theta(\lg V)$- Extract min
  - $\Theta(1)$ – decrease key
  - Total = $V^* \Theta(1) + V^*\Theta(\lg V) + E^*\Theta(1) = \Theta(E + V \lg V)$

Time complexity – summary

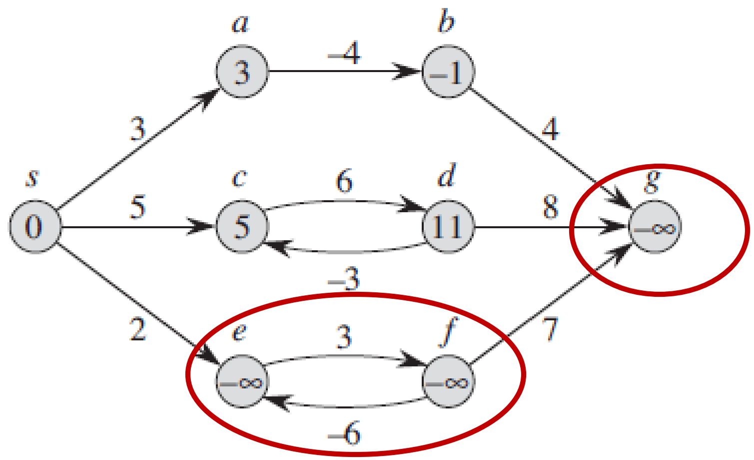
Data Structure used	Insert (V times)	Extract-min (V times)	Decrease key(E times)	Total
Array	$\Theta(1)$	$\Theta(V)$	$\Theta(1)$	$\Theta(V^2)$
Binary Heap	$\Theta(\lg V)$	$\Theta(\lg V)$	$\Theta(\lg V)$	$\Theta(E \lg V)$
Fibonacci Heap	$\Theta(1)$	$\Theta(\lg V)$	$\Theta(1)$	$\Theta(E + V \lg V)$

Negative weight vs. negative cycle

- If the graph G (V,E) contains no negative weight cycles reachable from the source s,
  - then for all vertices v ε V , the shortest-path weight δ(s, v) remains well defined, even if it has a negative value.
- If the graph contains a negative-weight cycle reachable from s
  - shortest-path weights are not well defined.

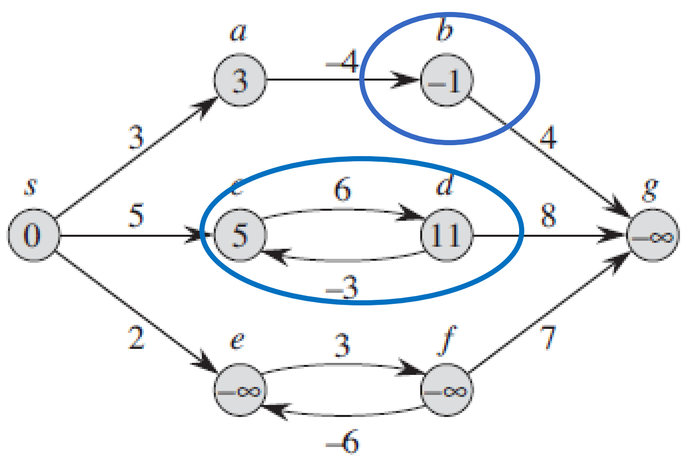
Negative weight vs. negative cycle

- Path weight of ***e*** and ***f*** continues to decrease as we follow the loop.
- As a result weight for ***g*** will also keep decreasing.
- So, path of these 3 nodes become undefined.



NEGATIVE WEIGHT VS. NEGATIVE CYCLE

- On the contrary even though path to **b** contains a negative weight but as there is no negative cycle, b will have a defined shortest path.
- Also **c** and **d** has a cycle but not negative cycle. So, they also have defined shortest path.



Bellman-Ford algorithm

Bellman-Ford algorithm

- The ***Bellman-Ford algorithm solves*** the general case in which edge weights may be negative.
- Given a weighted, directed Graph G(V,E) with source s and weight function w,
  - It returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source.
  - If there is such a cycle, the algorithm indicates that no solution exists.
  - If there is no such cycle, the algorithm produces the shortest paths and their weights.
  - The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

Bellman-Ford algorithm

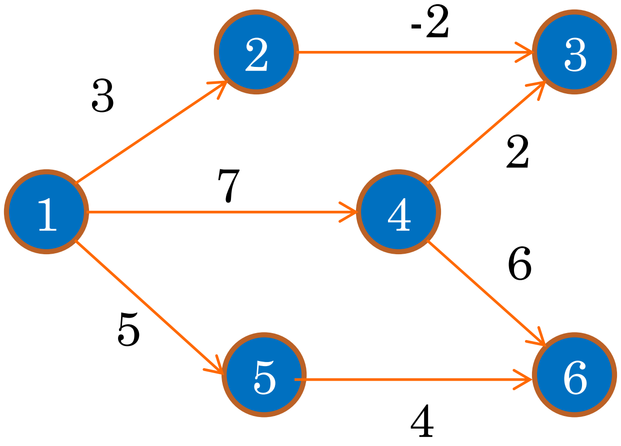
```
BELLMAN-FORD(G, w, s):
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
// The loop below is for checking if we can farther
// reduce the cost. And if so, there is a cycle
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

At each iteration, at least one vertex reachable from s will have the shortest path.

So, after V-1 iteration all V-1 vertices will have their shortest path

If we can farther reduce the cost then there is a cycle

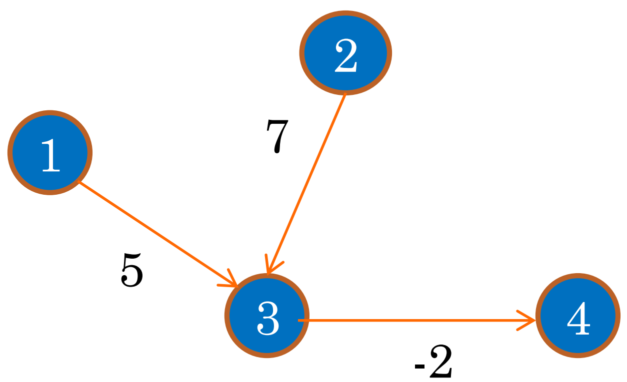
BELLMAN-FORD ALGORITHM - EXAMPLE



Process edges in following order.

5-6
4-6
4-3
2-3
1-2
1-4
1-5

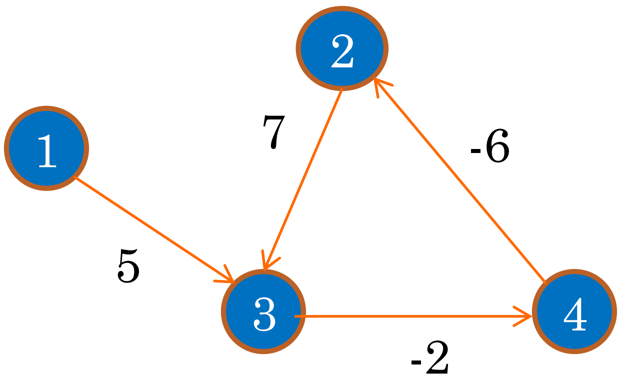
Bellman-Ford algorithm - Example



Process edges in following order.

3-4
2-3
1-3

Bellman-Ford algorithm - Example



Process edges in following order.

3-4
2-3
4-2
1-3

Try at home

- Increase all weights by a fixed number.
- How to convert single source Shortest Path to a Single Destination Shortest path algorithm.
- Apply Dijkstra to a graph with negative weight/cycle.

Reference

- Chapter 24 (Cormen) -> 24.1, 24.3