Graph Algorithms

Mohammad Javad Dousti

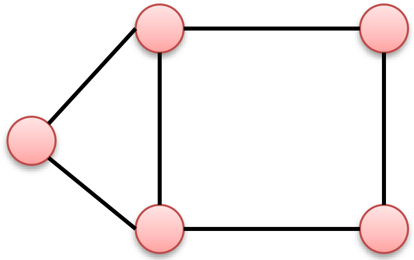
Overview

- Minimum Spanning Tree (MST)
  - Kruskal's Algorithm
  - Prim's Algorithm
- Shortest Path Problem
  - Importance of shortest path
  - Various versions of shortest path problem
  - Dijkstra Algorithm
  - Bellman-Ford Algorithm
  - Floyd-Warshall Algorithm
- Sample Problems

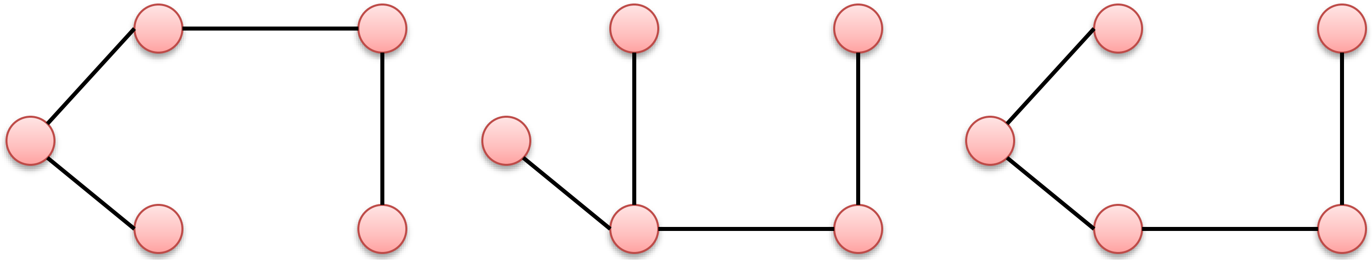
Minimum Spanning Tree

Basic Definitions

- **Spanning Tree:** A set of edges that connect all vertices of a graph while being a tree.



Graph *G*



Spanning Tree examples

- **Minimum Spanning Tree (MST):** A minimum-cost set of edges that connect all vertices of a graph.

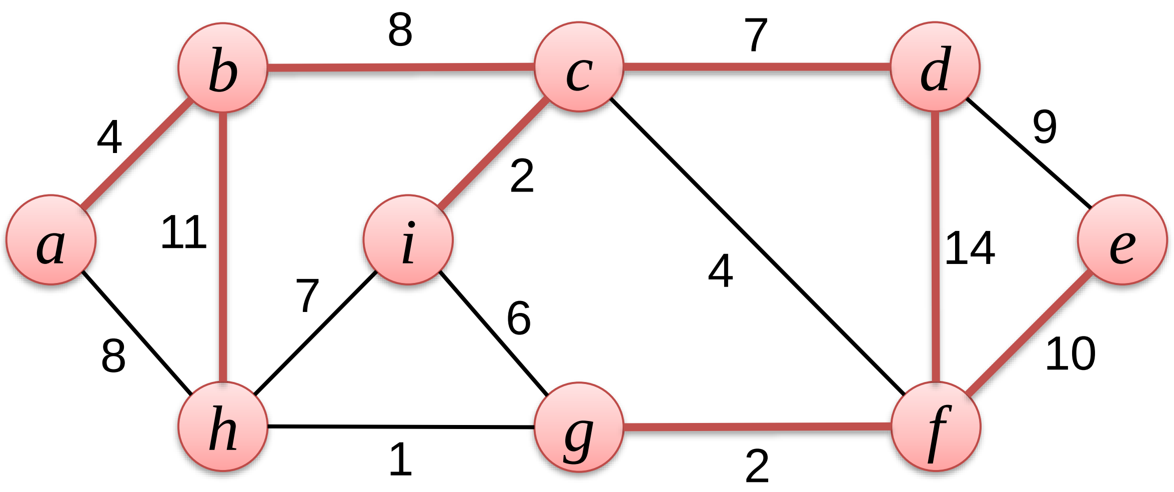
Problem Definition

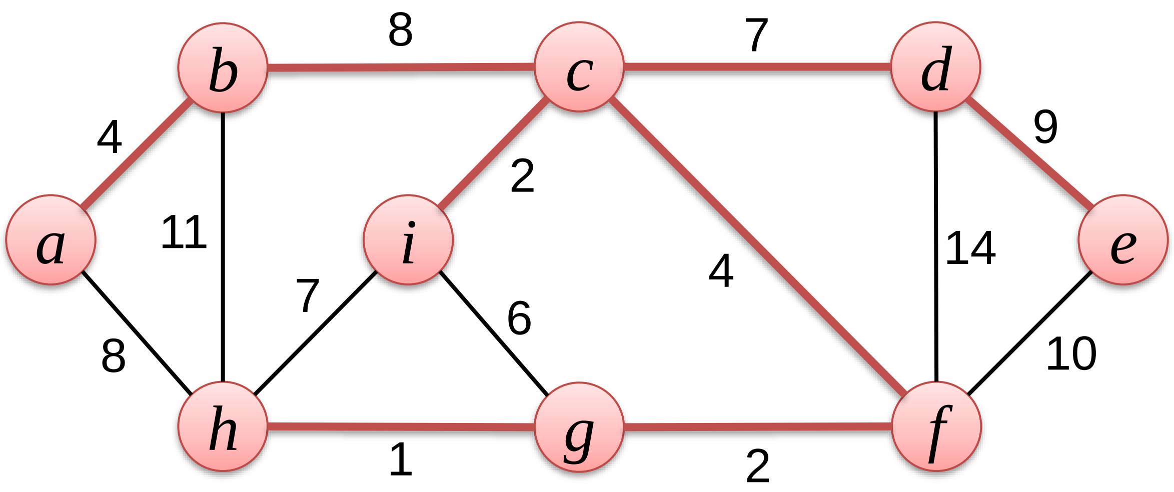
- **Given:** A weighted and connected graph $G = (V,E)$, where weights of each edge is defined as $w(e)$.
- Cost of a spanning tree T is defined as:

$$Cost(T) = \Sigma_{e \in T} w(e)$$

- **Goal:** Find a spanning tree of graph G such that its cost is minimized (i.e., find a *minimum spanning tree*.)

MST Example



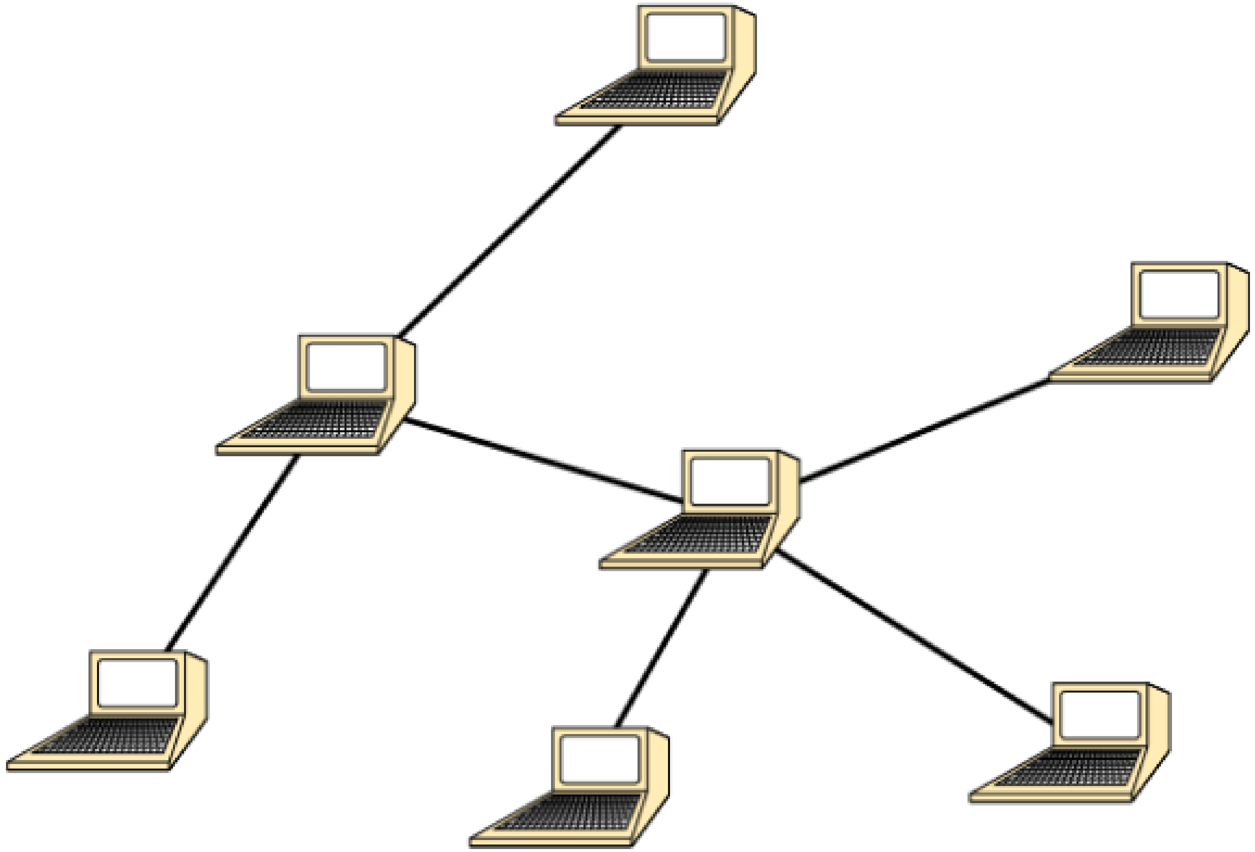


- $Cost(T) = 58$
- Is this spanning tree an MST?

- $Cost(T) = 37$
- Is MST unique?
  - No, one can remove (b,c) and add (a,h) and still have an MST.

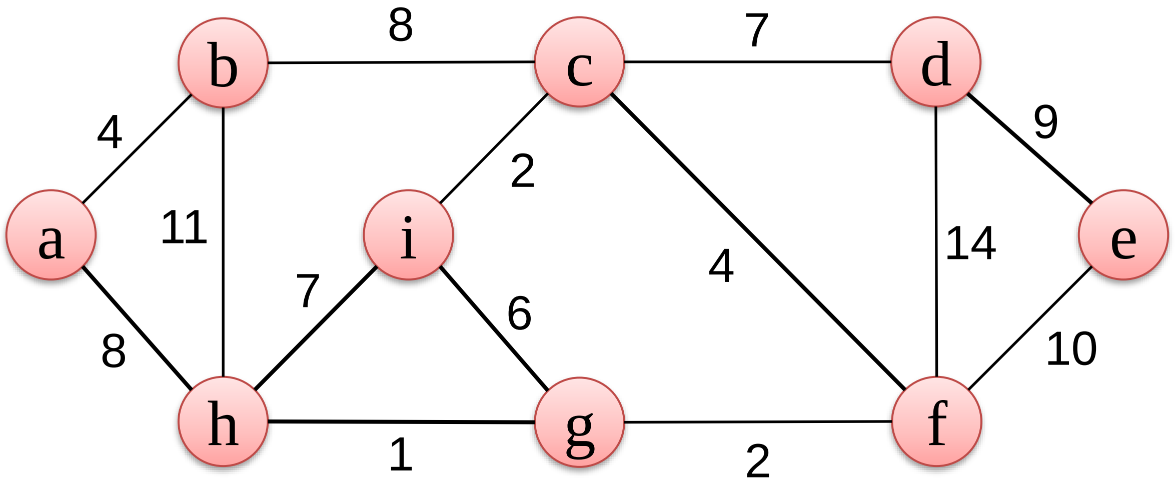
Applications

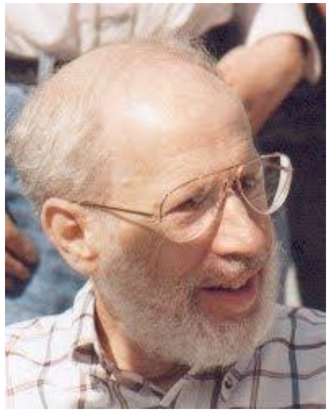
- Connect nodes in the least expensive way (i.e., with minimum pieces of wire):
  - Networking
  - Circuit design



Kruskal’s Algorithm Idea

- **Greedy choice:** Pick an edge with the lowest weight such that it doesn’t create a cycle.
  - Is there any other greedy choice?
- Example:



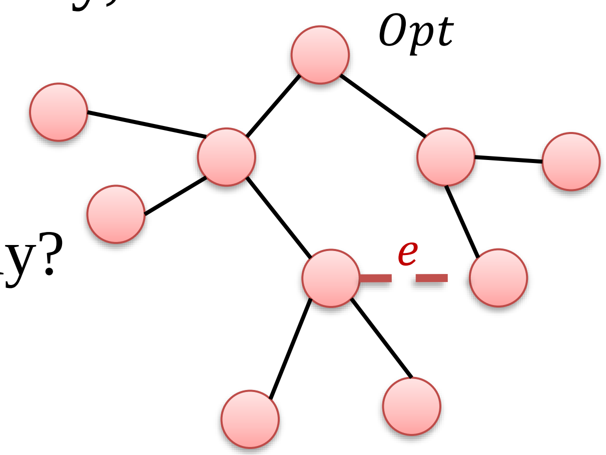


Joseph Kruskal

Edge	h-g	g-f	c-i	a-b	c-f	i-g	c-d	i-h	b-c	a-h	d-e	e-f	b-h	d-f
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14

Proof of Optimality

- Suppose Opt is an optimal solution which resembles the most to the greedy solution. We have $Cost(Opt) < Cost(greedy)$. Otherwise, we are done.
- Goal: We want to create another solution called Opt' such that:
  - $Cost(Opt') \leq Cost(Opt)$
  - It is more similar to the greedy solution.
- Sort the edges in the graph by their weights.
- Next, pick the first edge e which exists in either Opt and greedy, but not both.
  - $e \in greedy$ and $e \notin Opt$. Why?
  - $Opt + e$ has a cycle. Why?
  - There is another edge, e', in this cycle such that $w(e') \geq w(e)$. Why?



Proof of Optimality (cont'd)

- Now consider $Opt' = Opt - e' + e$.
  - $Cost(Opt') \leq Cost(Opt)$
  - Opt' is an MST.
  - Opt' resembles the greedy solution more than Opt, which is a contradiction. Hence, greedy solution is optimal.

Implementation Details

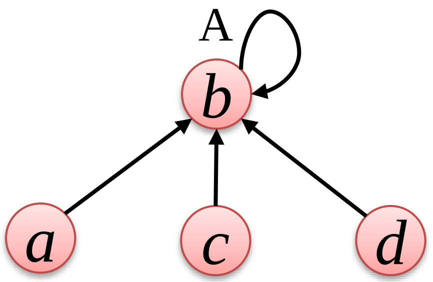
- Sorting edges with their weights takes $\mathcal{O}(|E|\log|E|)$.
- Each time we pick an edge, we should ensure that it doesn't make any cycle.
  - This process repeats $|E|$ times. Why not $|V|$ times?
  - We need a data structure to perform this check efficiently:

Disjoint–Set or Union–Find Data Structure

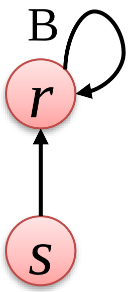
- Suppose we have several disjoint sets and we'd like to perform the following operations well:
  - Union: Given two sets, combine these two sets.
  - Find: Given an element, return the set which the element belongs to.
- Example:
  - $X = \{a, b, c\}$
  - $Y = \{d, e, f\}$
  - $Z = \{g\}$
  - `union(X, Z)` $\rightarrow \{a, b, c, g\}, \{d, e, f\}$
  - `find(d)` $\rightarrow Y$

Disjoint-Set Implementation

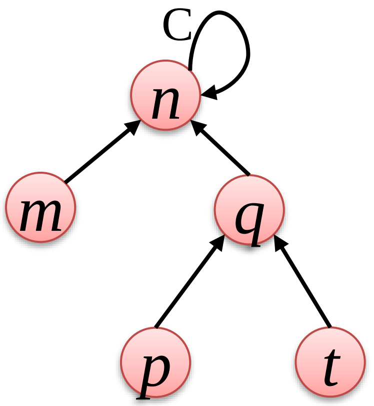
- Each disjoint set can be represented with a directed tree, where each node has a parent.



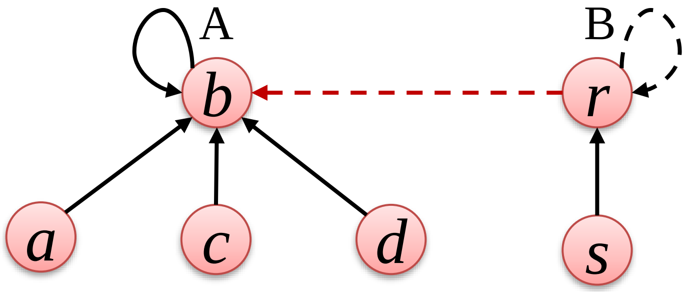
{a, **b**, c, d}



{**r**, s}



{m, **n**, p, q, t}



union(b, r) = {a, **b**, c, d, r, s}

```
find(u) {
    while parent(u) ≠ u
        u = parent(u)
    return u
}
```

```
union(a, b) {
    r1 = find(a)
    r2 = find(b)
    if r1 == r2 {
        return  // do nothing
    }
    parent(r1) = r2
}
```

Disjoint-Set Implementation Improvement

- Each tree holds height. During the union, tree with higher height becomes the parent.
  - This change causes a tree with n nodes to have height of $\mathcal{O}(\log n)$.

```
union(a, b){
    r₁ = find(a)
    r₂ = find(b)
    if r₁ == r₂ {
        return  // do nothing
    }
    if h[r₁] > h[r₂] {
        parent[r₂] = r₁
    } else if h[r₁] = h[r₂] {
        parent[r₁] = r₂
        h[r₂] = h[r₂] + 1
    } else {
        parent[r₁] = r₂
    }
}
```

  - A node r which is root of a subtree with height h has at least 2^h nodes.
    - Note the height of a tree increases only when $h[r_1] = h[r_2]$.
    - Use induction to prove.

Disjoint-Set Runtime Complexity Analysis

- Runtime complexity of `find(u)`
  - The height of the tree: $\mathcal{O}(h[u]) = \mathcal{O}(\log n)$
- Runtime complexity of `union(a,b)`
  - Two calls to `find(a)` and `find(b)`: $\mathcal{O}(max\{h[a], h[b]\}) = \mathcal{O}(\log n)$
  - In the best case, a and b are both roots of their respective trees and the operation takes $\mathcal{O}(1)$.

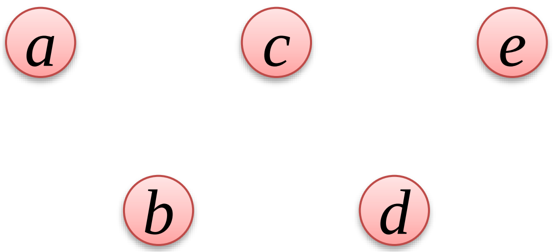
Kruskal’s Algorithm

```
Kruskal(G, w) {
    T = {}
    for each vertex v ∈ G.V {
        MakeSet(v)
    }
    E′ = sort(G.E, w)
    for each edge e=(u, v) ∈ E′ {
        r_u = find(u)
        r_v = find(v)
        if r_u ≠ r_v {
            T = T ∪ {e}
            union(r_u, r_v)
        }
    }
    return T
}
```

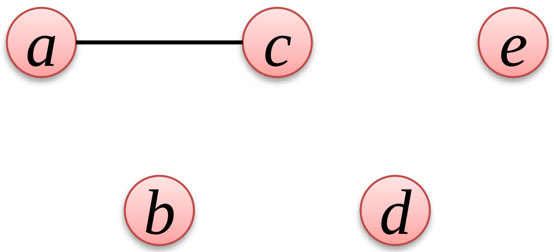
Runtime Complexity?

$$\underbrace{\mathcal{O}(|E|\log|E|)}_{\text{sort}} + \underbrace{\mathcal{O}(|E|\log|V|)}_{\text{for-loop}} = \mathcal{O}(|E|\log|V|)$$

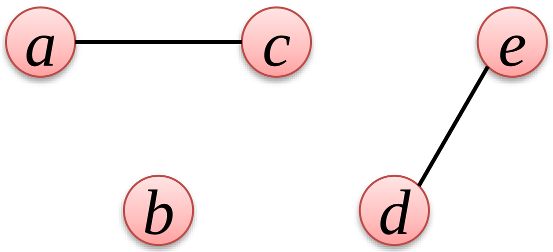
Kruskal Example w/ Disjoint Sets



{a}
{b}
{c}
{d}
{e}

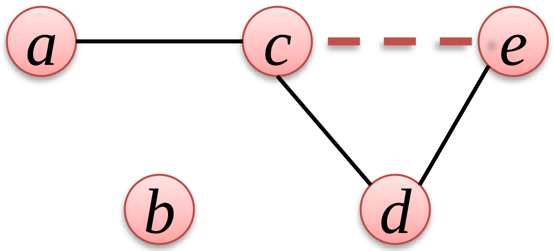


{a,c}
{b}
{d}
{e}



{a,c}
{b}
{d, e}

Creates a cycle: *c* and *e* are already in the same set

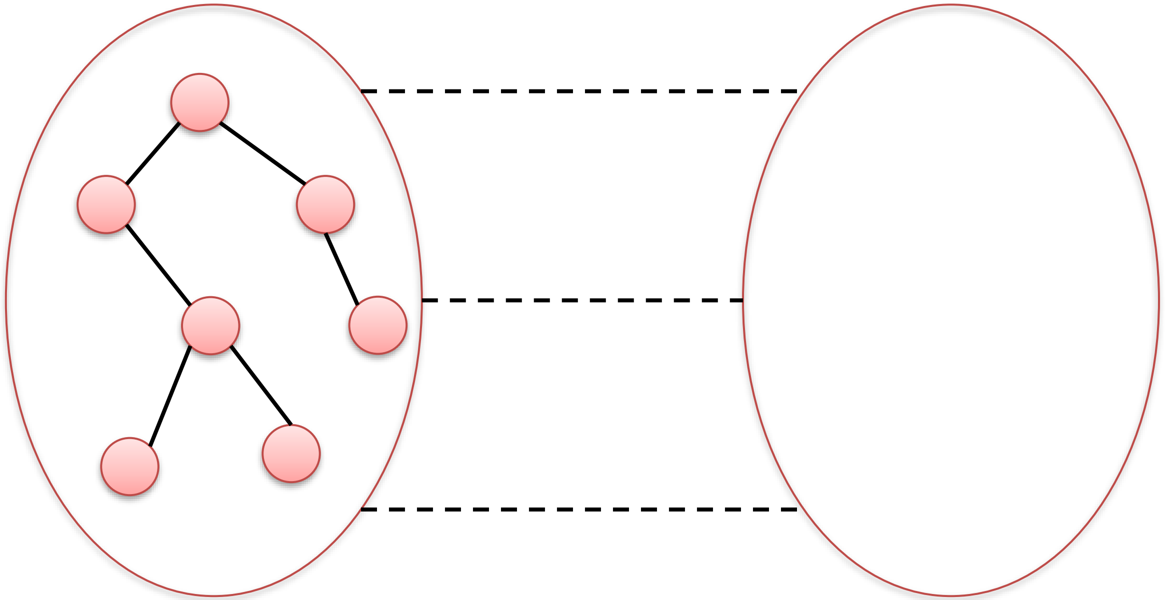


{a, c, d, e}
{b}

Prim's Algorithm Idea

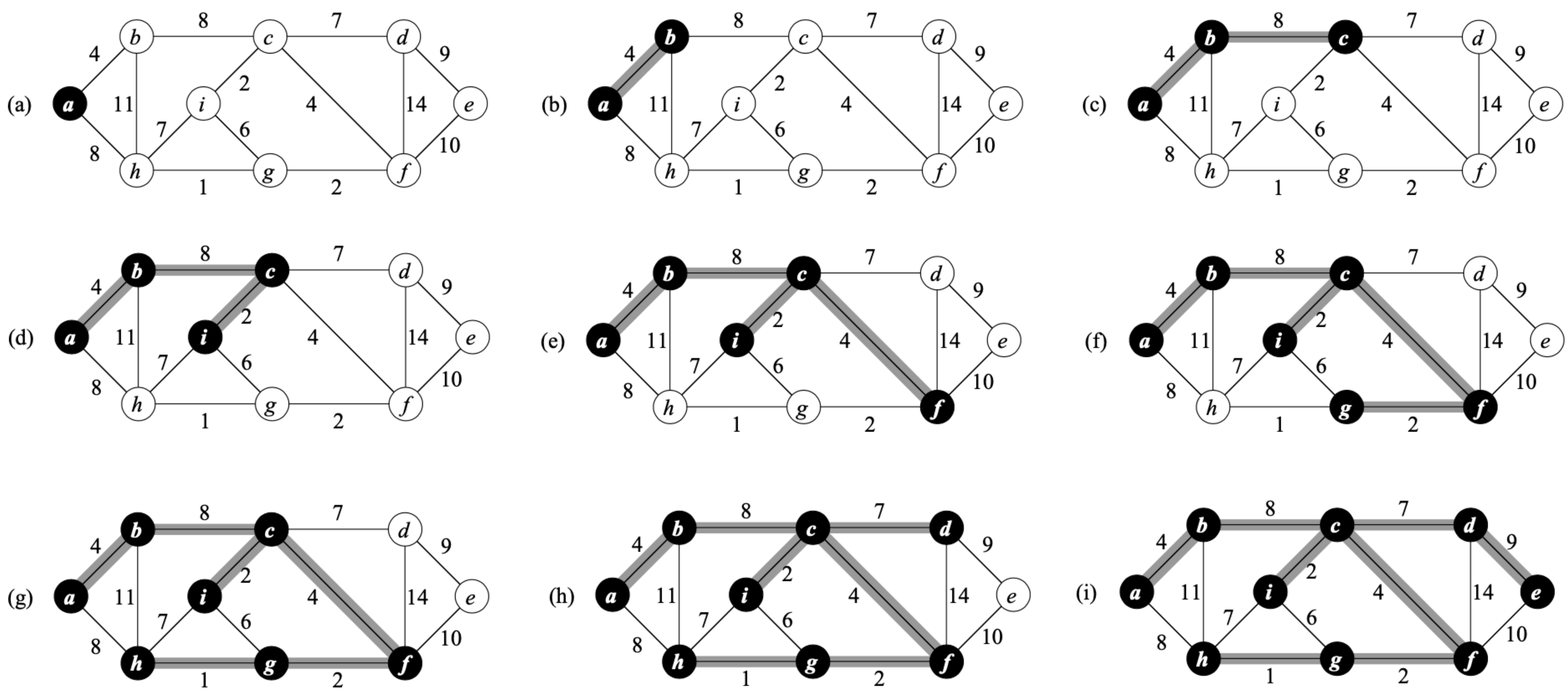
- **Greedy choice:** Choose the *best* edge connecting S to $V-S$.

Robert C. Prim



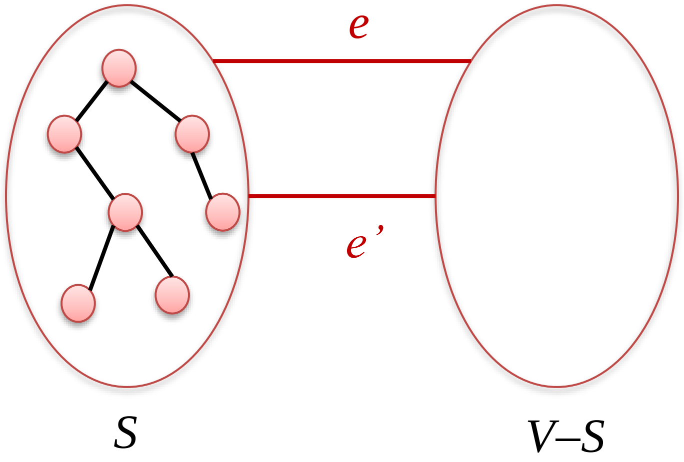

```
Prim(G, w){
    T = {}
    S = {a} // pick an arbitrary node in G
    while |S|<|V| {
        (u,v) = e = argmin_{u∈S, v∈V−S} {w(u, v)}
        S = S + {v}
        T = T + {e}
    }
    return T
}
```

Example



Proof of Optimality

- Proof is very similar to that of Kruskal's algorithm.
- Consider Opt as the most similar optimal solution to the greedy one.
- We create Opt' such that $Cost(Opt') \leq Cost(Opt)$ and be more similar to the greedy solution than Opt.
- Consider edge e as the first difference between greedy solution and Opt.
  - $e \in greedy$ and $e \notin Opt$.
  - $Opt + e$ has a cycle.
  - There is another edge, e', in this cycle such that $w(e') \geq w(e)$.
- $Opt' = Opt - e' + e$
  - $Cost(Opt') \leq Cost(Opt)$



Prim's Algorithm: Approach 1

```
Prim(G, w){
    H = {}
    S = {a} // pick an arbitrary node in G
    for each u as neighbor of a
        AddToHeap(H, (a, u), w(a, u))
    T = {}
    while |S| < |V| {
        (u, v) = Extract-Min(H)
        if u ∉ S or v ∉ S {
            if u ∉ S {
                swap(u, v)
            }
            // u ∈ S, v ∉ S
            T = T + {(u, v)}
            S = S + {v}
            for each x as neighbor of v {
                if x ∉ S
                    AddToHeap(H, (v, x), w(v, x))
            }
        }
    }
    return T
}
```

$\boxed{\mathcal{O}(\log|E|)}$ (AddToHeap, initial loop)

$\boxed{\mathcal{O}(\log|E|)}$ (Extract-Min)

$\boxed{\mathcal{O}(\log|E|)}$ (AddToHeap, inner loop)

Runtime Complexity: $\mathcal{O}(|V|\log|E| + |E|\log|E|) = \mathcal{O}(|E|\log|V|)$

Prim's Algorithm: Approach 2

- Idea: For each $v \notin S$, we keep the lowest cost it takes to connect v to S.

Runtime Complexity: $\mathcal{O}(|V|^2)$

Can this be improved?

```
Prim(G, w){
    S = {a} // pick an arbitrary node in G
    T = {}
    Initialize minW[] for all nodes to +∞
    for each u as neighbor of a {
        minW[u] = w(a, u)
        parent[u] = a
    }
    while |S| < |V| {
        u = argmin_{v∉S}{minW[v]}
        S = S + {u}
        T = T + (parent[u], u)
        for each x as neighbor of u {
            if x ∉ S and minW[x] > w(u, x) {
                minW[x] = w(u, x)
                parent[x] = u
            }
        }
    }
    return T
}
```

Summary: Kruskal vs. Prim

- Both are Greedy algorithms
  - Both take the next minimum edge
  - Both are optimal (find the global min)
- Different sets of edges considered
  - Kruskal – all edges
  - Prim – Edges from Tree nodes to rest of G.
- Both need to check for cycles
  - Kruskal – set containment and union.
  - Prim – Simple boolean.
- Both can terminate early
  - Kruskal – when $|V| - 1$ edges are added.
  - Prim – when $|V|$ nodes are added (or $|V| - 1$ edges).

Shortest Path Algorithms

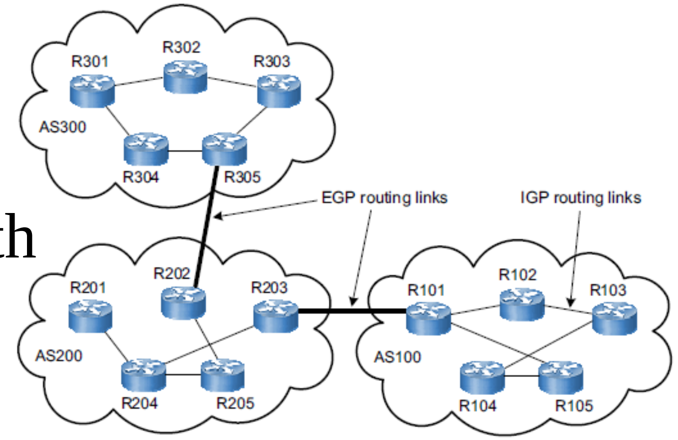
Some slides are courtesy of Dr. Mahini.

Overview

- Shortest Path Problem
  - Importance of shortest path
  - Various version of shortest path problem
  - Dijkstra Algorithm
  - Bellman-Ford Algorithm
  - Floyd-Warshall Algorithm

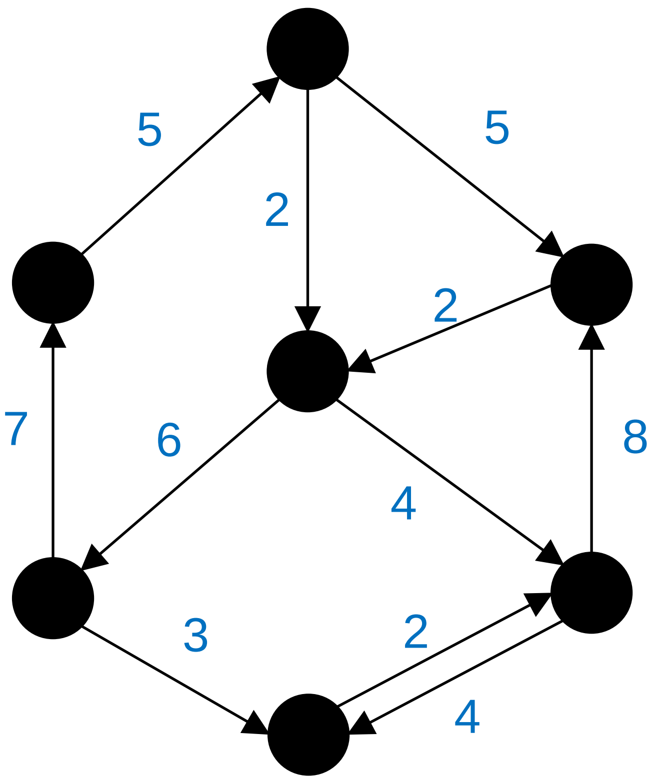
Applications of "Shortest Path" Problem

- Shortest path first (SPF) is used in the network routing protocols.
  - Routing: A protocol that specifies how routers communicate with each other, disseminating information that enables them to select routes between any two nodes on a computer network.

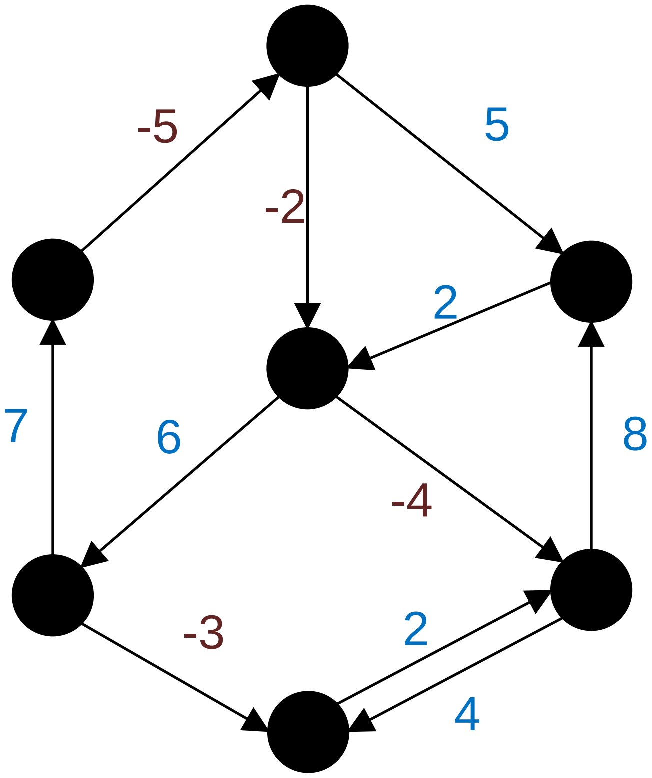


- GPS navigating systems
  - For a given source vertex (node) in the graph, the algorithm can be used to find shortest path <u>from a single starting vertex to a single destination vertex.</u>
  - If vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, shortest path algorithms can be used to find the shortest route between a city and another destination city.

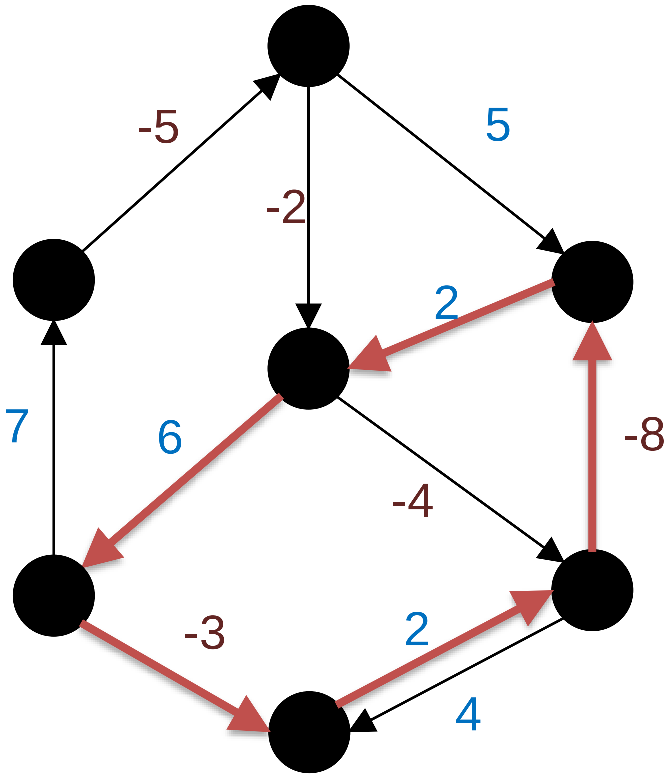
Shortest Path Problem Variants



Non-negative weights

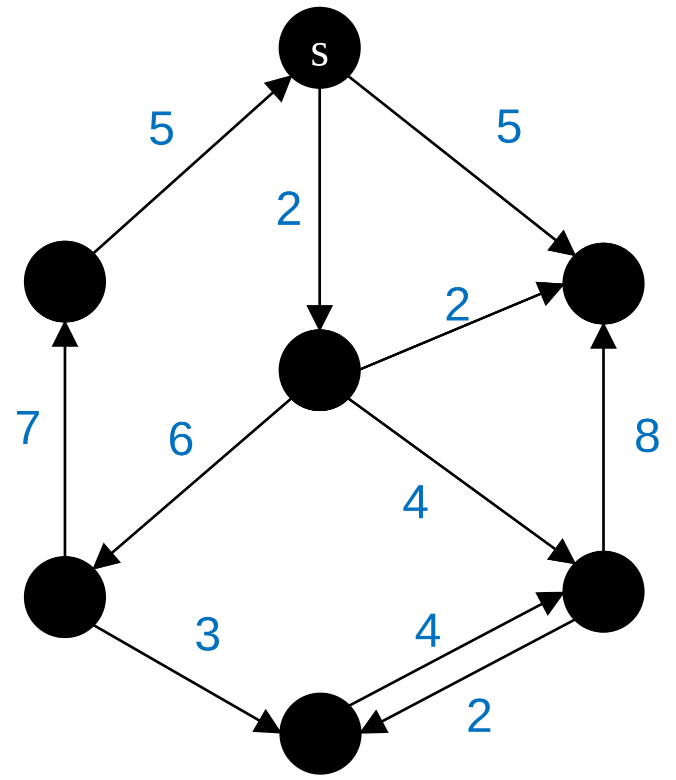


Negative weights
No negative cycle



Negative cycle

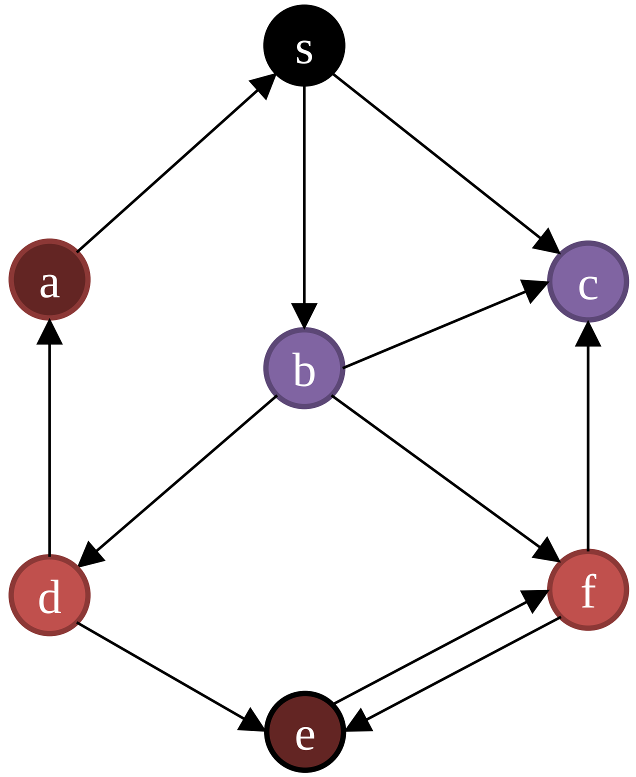
Shortest Path Problem Variants (cont'd)



Input: A directed weighted graph $G = (V, E)$. Assume $w(e)$ is the weight of edge $e \in E$.

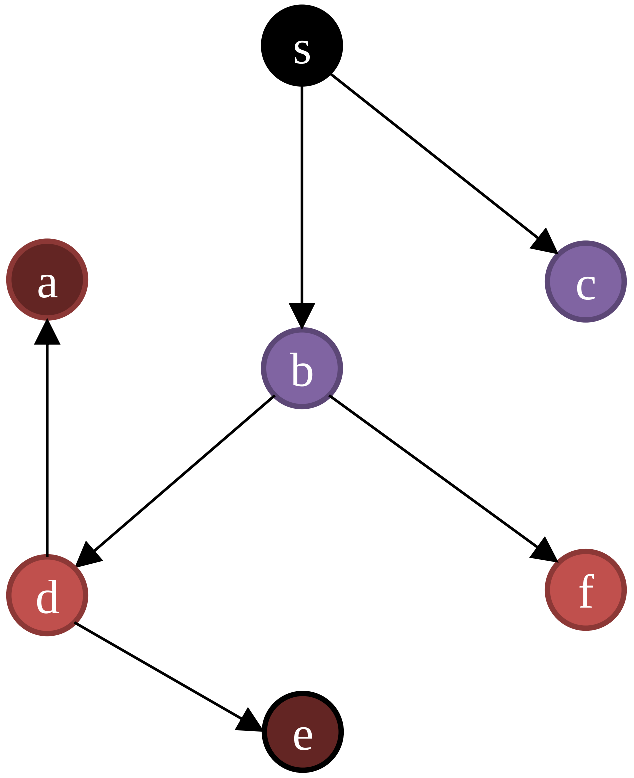
- Single Source: Goal is to find the shortest path form a given single source(s) to all other vertices
- All Pairs: Goal is to find the shortest path between any two vertices in the given graph

BFS Tree For a Non-Weighted Graph

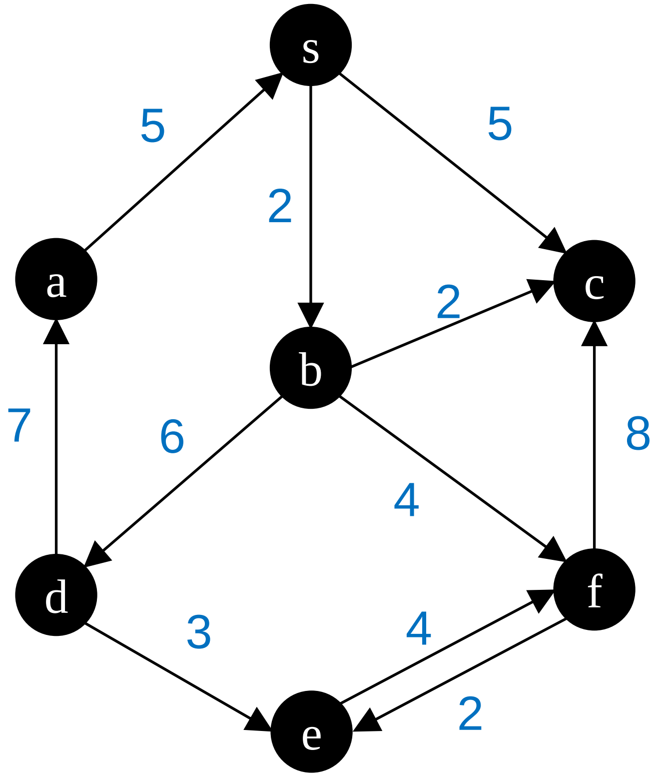


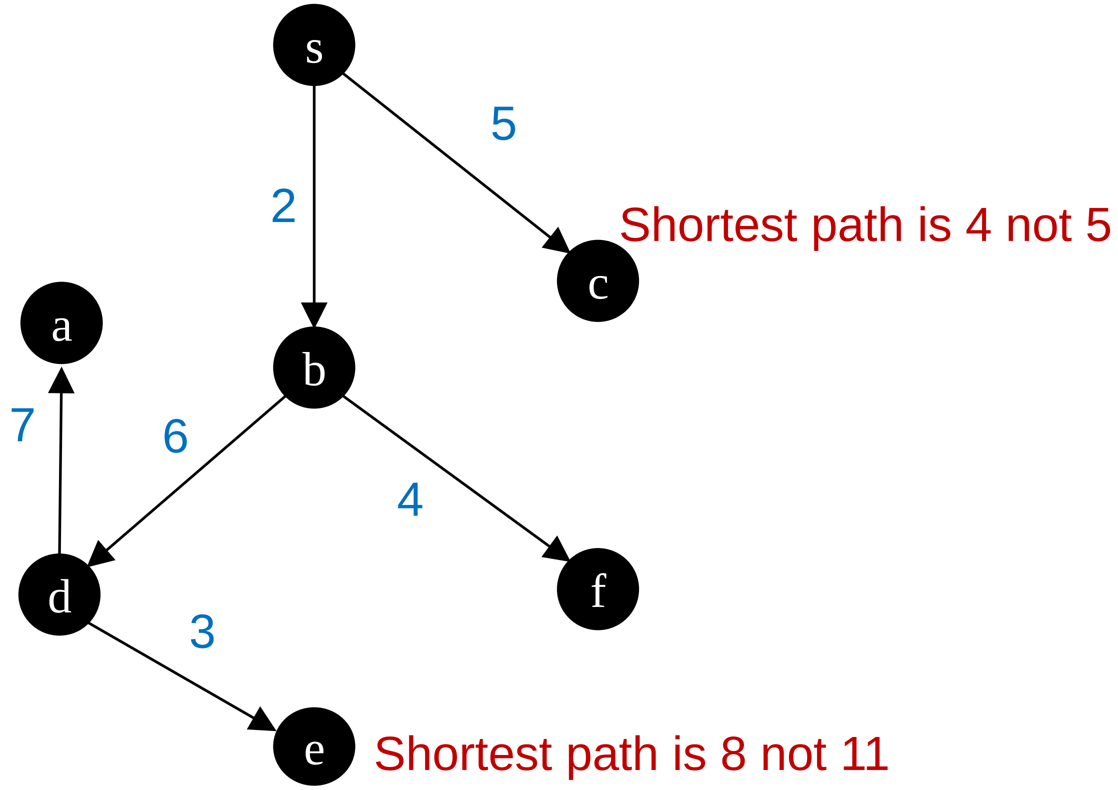
Imagine BFS on this graph form node *s*

order	0	1	2	3	4	5	6
node	s	b	c	d	f	a	e
distance	0	1	1	2	2	3	3



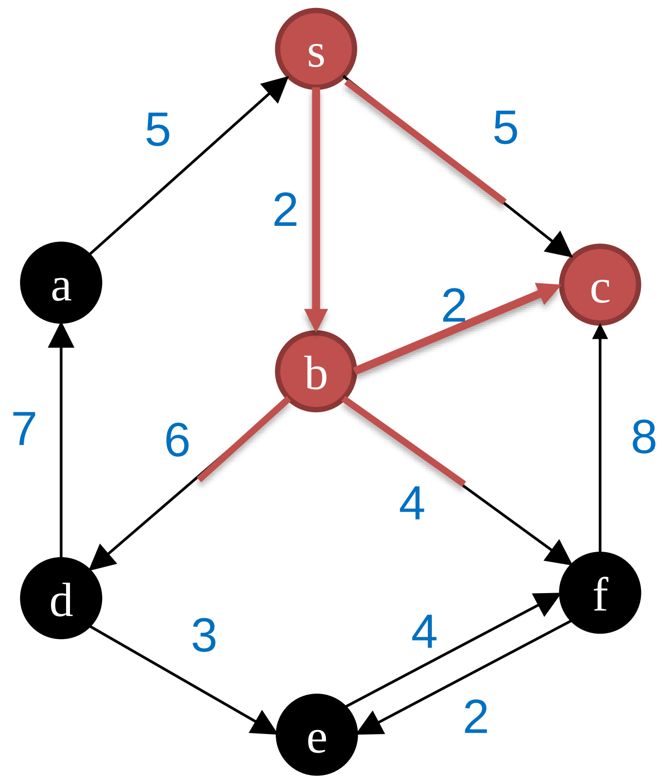
BFS Tree For a Weighted Graph





BFS Tree

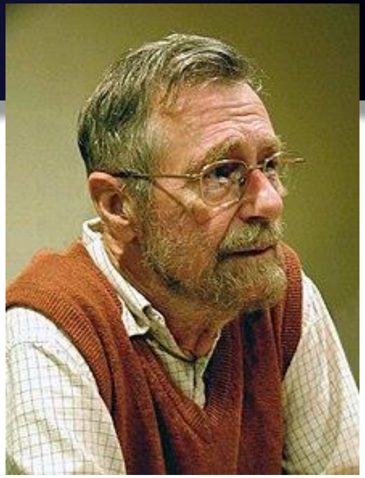
How can we use the same idea as BFS?



Dijkstra's Algorithm

Some slides are courtesy of Dr. Mahini.

Dijkstra's Algorithm



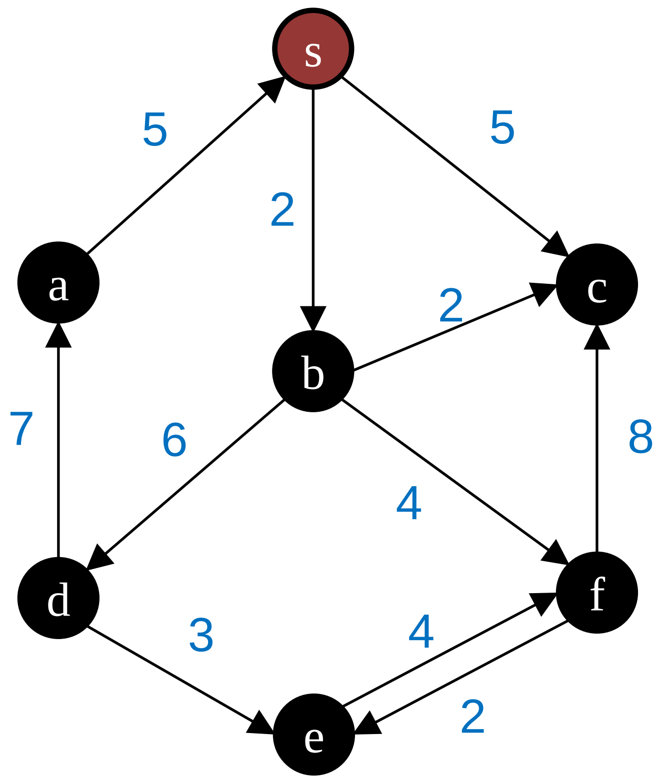
Edsger Dijkstra

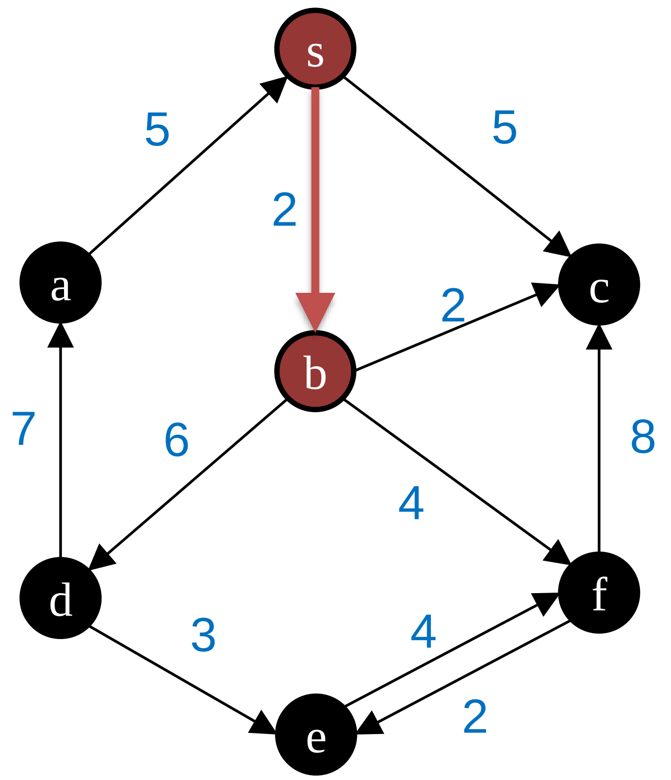
- ❑ "*Dijkstra*" is pronounced as /ˈdaɪkstrə/ or /dike·struh/.
- ❑ Non-negative weights
- ❑ Single Source
- ❑ **Definitions**
  - ➢ Set S: Set of all nodes that their shortest path are found.
  - ➢ $dist[v]$: minimum distance of node v from source s, such that all middle nodes are in set S.

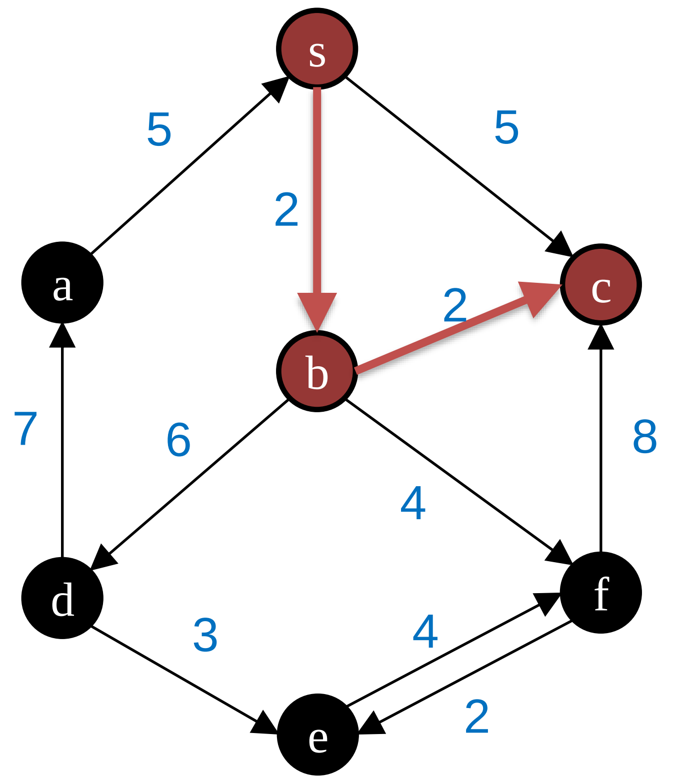
Algorithm
- Start from $S = \{\}$, and $dist[s] = 0$, $dist[v] = \infty$
- In each step do the followings:
  - Find $v \in V - S$ with minimum value of $dist[v]$. Let's name it u.
  - Add u to set S.
  - Update value of array $dist$. In particular, for each $v \in V - S$ let $dist[v] = \min\{dist[v], dist[u] + w(u, v)\}$.

Greedy choice

Example





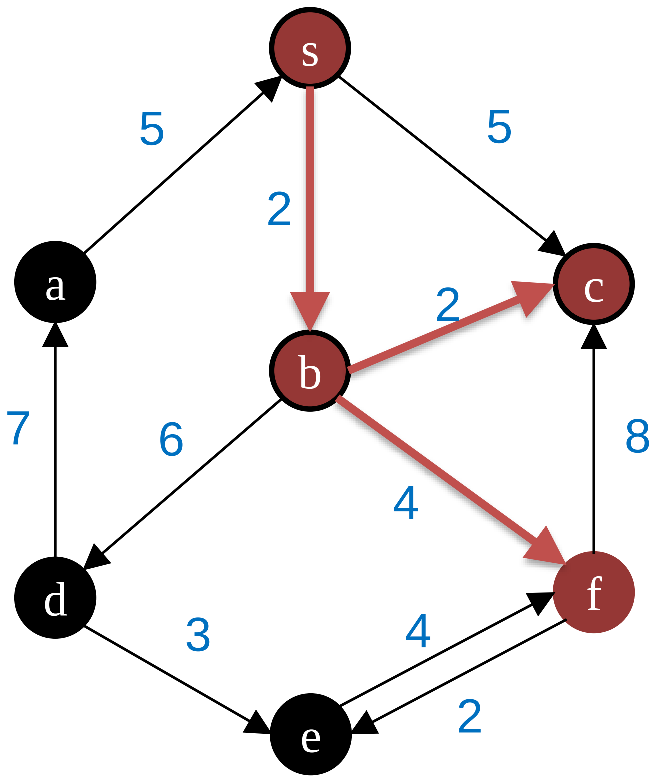


node	s	a	b	c	d	e	f
dist	0	∞	2	5	∞	∞	∞

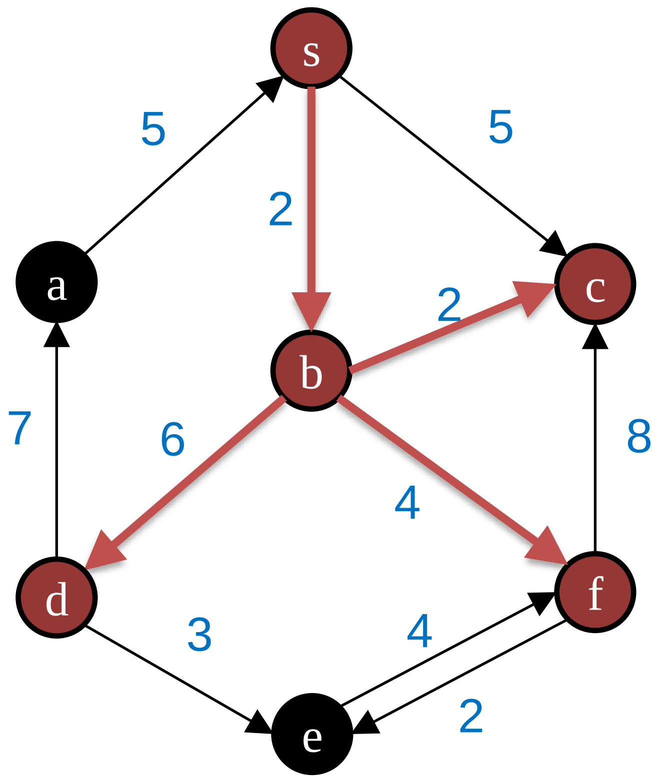
node	s	a	b	c	d	e	f
dist	0	∞	2	4	8	∞	6

node	s	a	b	c	d	e	f
dist	0	∞	2	4	8	∞	6

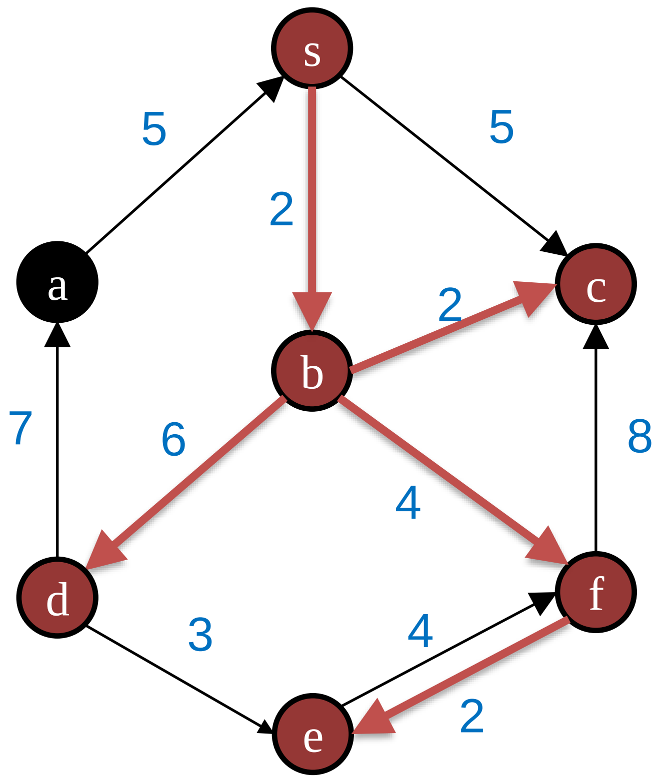
Example (cont'd)



node	s	a	b	c	d	e	f
dist	0	∞	2	4	8	8	6

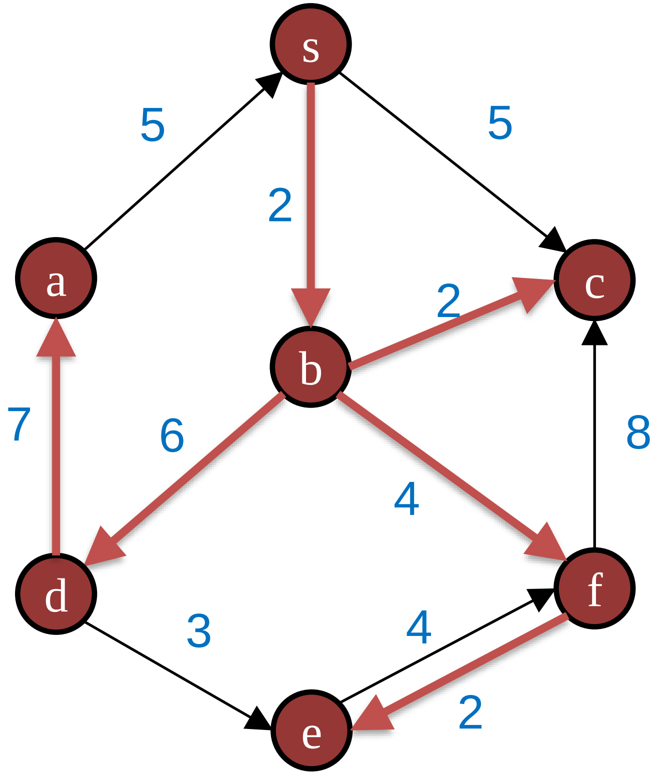


node	s	a	b	c	d	e	f
dist	0	15	2	4	8	8	6

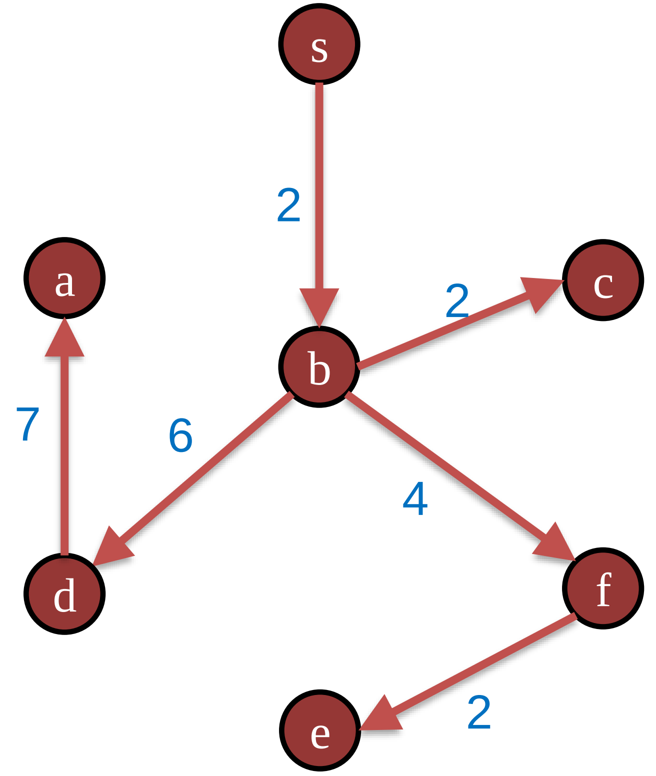


node	s	a	b	c	d	e	f
dist	0	15	2	4	8	8	6

Example (cont'd 2)



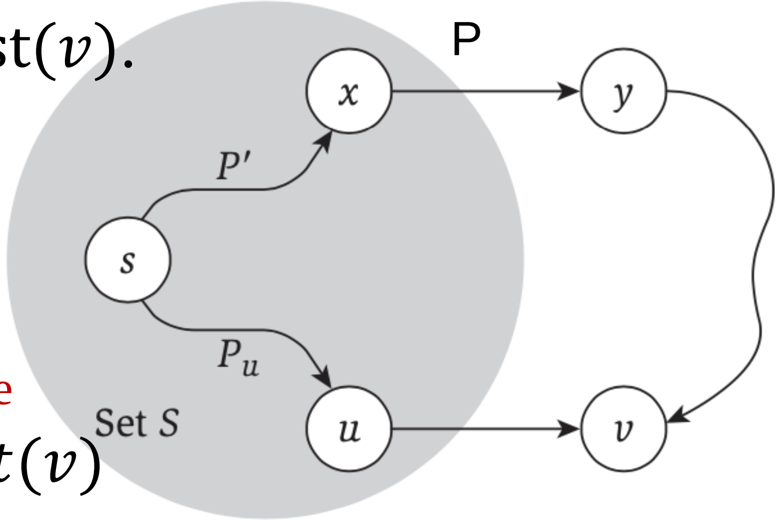
node	s	a	b	c	d	e	f
dist	0	15	2	4	8	8	6



Shortest Path Tree

Proof by Induction

- For each node $u \in S$, $dist[u]$ is the length of the shortest s-u path.
- **Base case:** $|S| = 1$ is trivial
- **Inductive hypothesis:** Assume true for $|S| = k \geq 1$.
  - Let *v* be the next node added to *S*, and let *u*-*v* be the chosen edge.
  - The shortest *s*-*u* path plus (*u*, *v*) is an *s*-*v* path of length dist(v).
  - Consider any *s*-*v* path *P*. We'll see that is no shorter than dist(v).
  - Let *x*–*y* be the first edge in *P* that leaves *S*, and let P' be the subpath to *x*.
  - *P* is already too long as soon as it leaves S and path *y*-*v* has positive weight.



$$l(P) \geq l(P') + w(x,y) \geq dist[x] + w(x,y) \geq dist(y) \geq dist(v)$$

Non-negative weights Induction Definition of *dist*(*y*) Greedy choice

Implementation 1: $O(|V|^2)$

```
Dijkstra(G, w, s) {
    S = {}
    for each node v ∈ V {
        dist[v] = ∞
        parent[v] = null
    }
    dist[s] = 0

    while |S|≠|V| {
        u = argmin_{v∈V−S} {dist[v]}
        S = S + {u}
        for each neighbor of u in v ∈ V - S
            if dist[v] > dist[u] + w(u, v){
                dist[v] = dist[u] + w(u, v)
                parent[v] = u
            }
    }
}
```

$\mathcal{O}(|V|)$

$\mathcal{O}(|V|)$

$\mathcal{O}(|E|)$

$\mathcal{O}(1)$

Runtime complexity: $\mathcal{O}(|E| + |V|^2) = \mathcal{O}(|V|^2)$

Implementation 2: $O(|E|\log|V|)$

```
Dijkstra(G, w, s) {
    S = {}
    for each node v ∈ V
        dist[v] = ∞
        parent[v] = null
    dist[s] = 0

    for each node v ∈ V
        Add v to Q with priority dist[v] **

    while Q ≠ {} {
        u = ExtractMin(Q) **
        S = S + {u}
        for each neighbor of u in V - S {  // O(d_u) or |E| in total
            if dist[v] > dist[u] + w(u, v) {
                dist[v] = dist[u] + w(u, v)
                parent[v] = u
                Decrease priority of v in Q with dist[v]
            }
        }
    }
}
```

Priority queue:

- Add (v, value)
- Extract min()
- Decrease(v, value)

$|V| \times \mathcal{O}(log|V|)$

$|V| \times \mathcal{O}(log|V|)$

$|E| \times \mathcal{O}(\log|V|)$

Heap review: https://www.hackerearth.com/practice/notes/heaps-and-priority-queues/

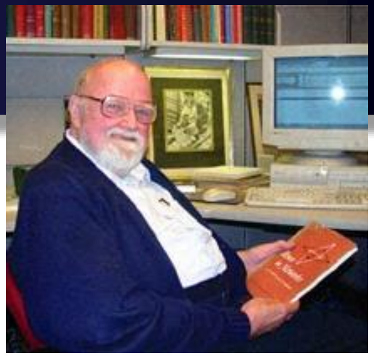
Bellman-Ford's Algorithm

Some slides are courtesy of Dr. Mahini.

Algorithm - $O(|V||E|)$

- Negative weights, No negative cycle
- Single Source
- Dynamic programming

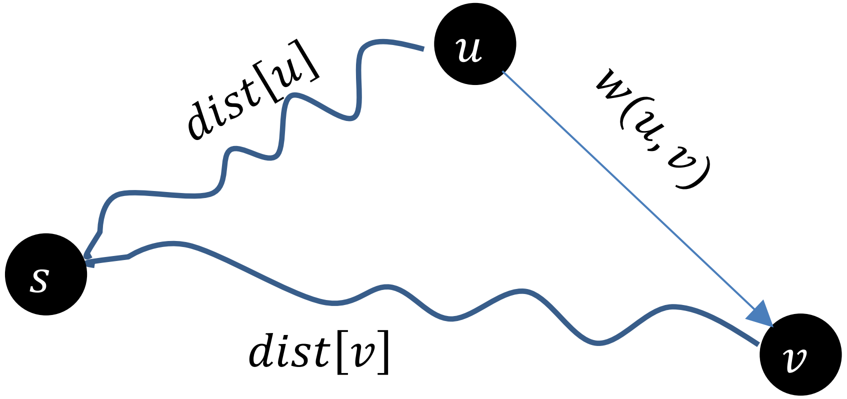
Richard E. Bellman

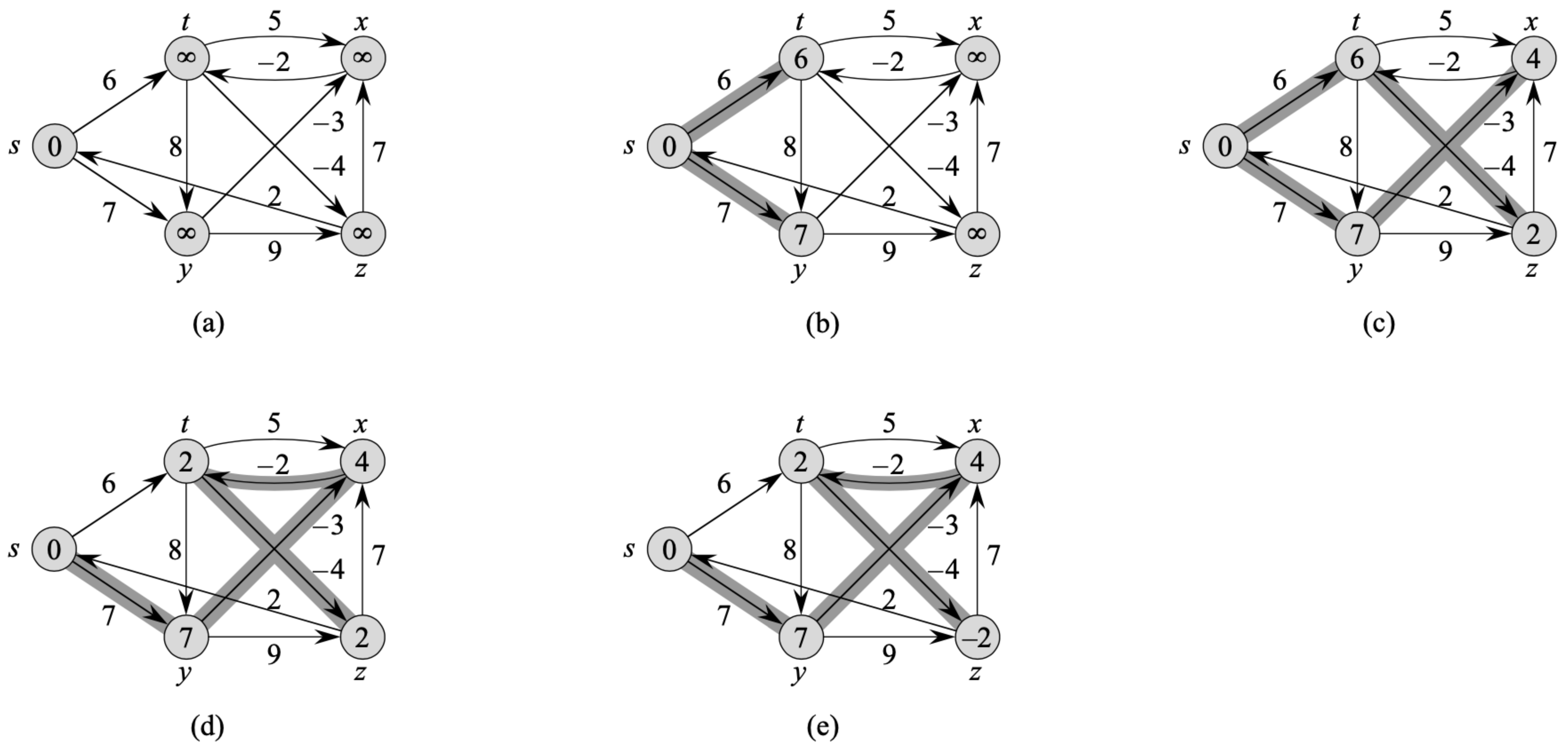


Lester R. Ford Jr.

```
Bellman-Ford(G, w, s) {
    for each node v ∈ V
        dist[v] = ∞
        parent[v] = null
    dist[s] = 0

    for i = 1 to |V| - 1
        for each edges (u, v) ∈ E
            Relax(u, v, w)
}
```

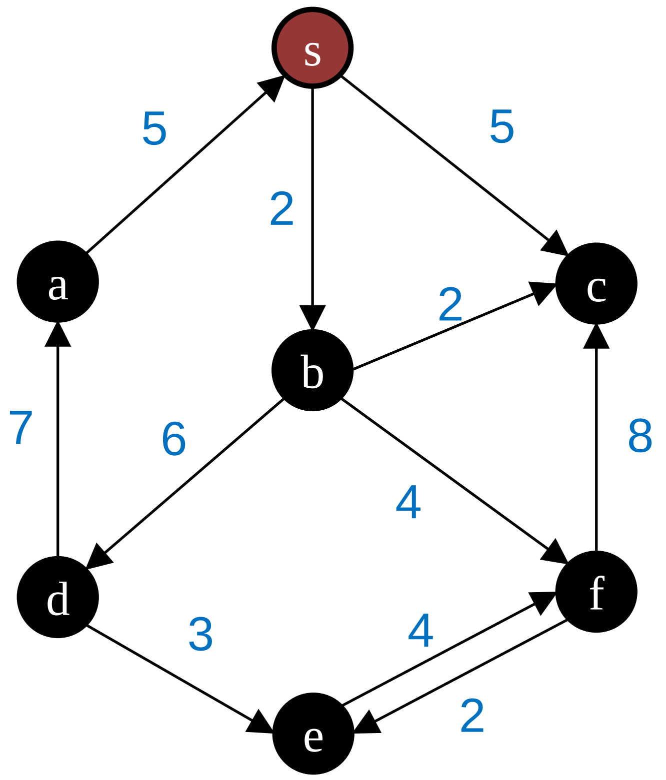
```
Relax(u, v, w){
    if dist[v] > dist[u] + w(u, v) {
        dist[v] = dist[u] + w(u,v)
        parent[v] = u
    }
}
```





t
x
s
y
z
5
−2
6
8
−3
−4
7
2
9
(a)
(b)
(c)
(d)
(e)

Example 2



#	node	s	a	b	c	d	e	f
0	dist	0	∞	∞	∞	∞	∞	∞
1	dist	0	∞	2	5	∞	∞	∞
2	dist	0	∞	2	4	8	∞	6
3	dist	0	15	2	4	8	8	6

Proof by Induction

Statements (assume $\delta[v]$ is the actual shortest path of node v):

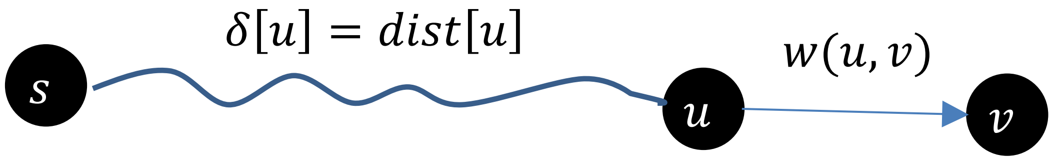
- For each node v such that its actual shortest path has k edges, we have $dist[v] = \delta[v]$ at the end of $i = k$ loop

Basis ($k = 0$)

- The statement is true for source node s

Assume the statement is true for k . We need to prove it for $k + 1$.

```
Bellman-Ford(G, w, s) {
    for each node v ∈ V
        dist[v] = ∞
        parent[v] = null
    dist[s] = 0

    for i = 1 to |V| - 1
        for each edges (u,v) ∈ E
            Relax(u, v, w)
}
```



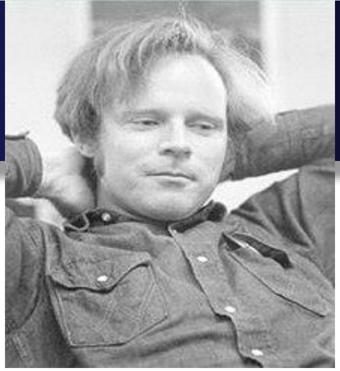
I) $\delta[v] = \delta[u] + w(u,v) = \text{dist}[u] + w(u,v) \geq dist[v]$

II) $dist[v] \geq \delta[v]$

Floyd-Warshall's Algorithm

Some slides are courtesy of Dr. Mahini.

Floyd-Warshall's Algorithm



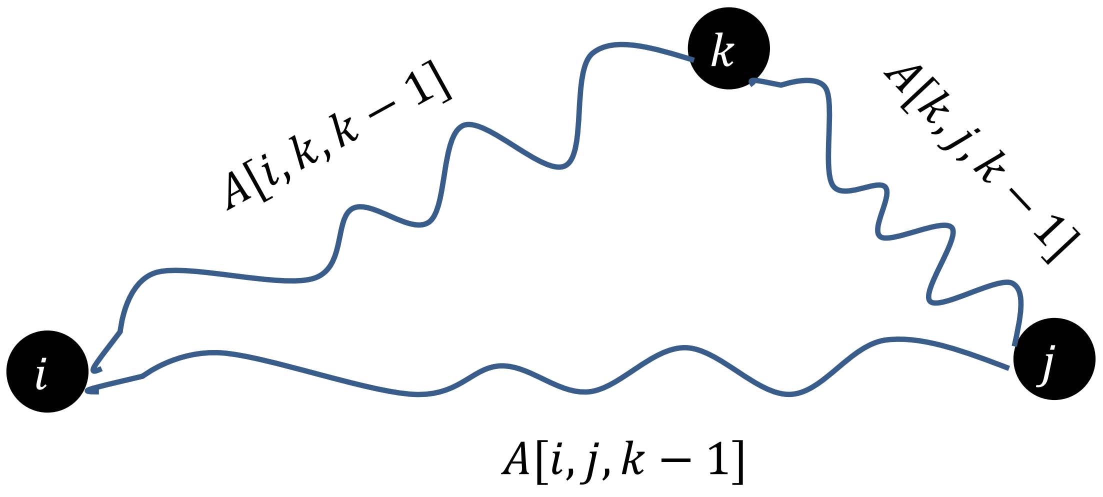
Robert W. Floyd

Stephen Warshall

- Negative weights, No negative cycle
- All pairs
  - The result is a matrix *A*, where element *A*[*i*,*j*] shows the shortest path between nodes *i* and *j* in the graph.
- Trivial method
  1. Use Dijkstra's algorithm |*V*| times for each node on the graph as source.
     - **Runtime complexity:** $\mathcal{O}(|V|^3)$ or $\mathcal{O}(|V||E|\log|V|)$.
     - Caveat: This method doesn't support graphs with negative weights.
  2. Use Bellman-Ford's algorithm |*V*| times for each node on the graph as source.
     - **Runtime complexity:** $\mathcal{O}(|V|^2|E|)$
     - For dense graphs (i.e., when $|E| = \mathcal{O}(|V|^2)$), runtime complexity becomes $\mathcal{O}(|V|^4)$.
     - Can we do any better?

Floyd-Warshall's Idea

- Floyd-Warshall algorithm uses dynamic programming.
- **Definitions**
  - $A[i,j,k]$: shortest path from node i to node j, such that all middle nodes have index less than or equal to k.
  - Solution will be stored in $A[i,j,n]$
  - $A[i,j,0] = w[i,j]$



$$A[i,j,k] = \min\{A[i,j,k-1], A[i,k,k-1] + A[k,j,k-1]\}$$

Implementation

```
Floyd-Warshall(W) {
    // W is an adjacency matrix
    n = W.rows
    for i = 1 to n
        for j = 1 to n
            A[i,j,0] = w[i,j]

    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1] + A[k,j,k-1])
    return A[:,:,n]
}
```

Runtime complexity: $\mathcal{O}(|V|^3)$

How to reduce the space/memory from $\mathcal{O}(|V|^3)$ to $\mathcal{O}(|V|^2)$?

Implementation with Better Memory

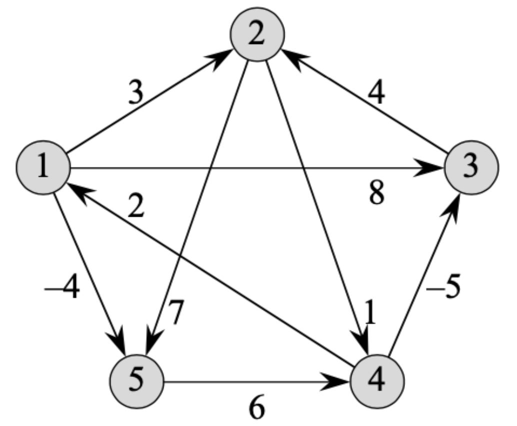
```
Floyd-Warshall(W) {
    // W is an adjacency matrix
    n = W.rows
    for i = 1 to n
        for j = 1 to n
            A[i,j] = w[i,j]

    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                A[i,j] = min(A[i,j], A[i,k] + A[k,j])
    return A
}
```

For all *i*, *j*, *k*

- `A[i, k, k-1] = A[i, k, k]`
- `A[k, j, k-1] = A[k, j, k]`

Space complexity: $\mathcal{O}(|V|2)$

Example



$$\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

`A[:,:,0]`

$$\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \underline{5} & -5 & 0 & \underline{-2} \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

`A[:,:,1]`

$$\begin{pmatrix} 0 & 3 & 8 & \underline{4} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \underline{5} & \underline{11} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

`A[:,:,2]`

$$\begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & \underline{-1} & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

`A[:,:,3]`

$$\begin{pmatrix} 0 & 3 & \underline{-1} & 4 & -4 \\ \underline{3} & 0 & \underline{-4} & 1 & \underline{-1} \\ \underline{7} & 4 & 0 & 5 & \underline{3} \\ 2 & -1 & -5 & 0 & -2 \\ \underline{8} & \underline{5} & \underline{1} & 6 & 0 \end{pmatrix}$$

`A[:,:,4]`

$$\begin{pmatrix} 0 & \underline{1} & \underline{-3} & \underline{2} & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

`A[:,:,5]`

Summary

Algorithm	Usage	Graph	Runtime
BFS/DFS	Minimum Spanning Tree	Unweighted	$\mathcal{O}(V+E)$
Kruskal	Minimum Spanning Tree	Weighted	$\mathcal{O}(E \log V)$
Prim	Minimum Spanning Tree	Weighted	$\mathcal{O}(E \log V)$ or $\mathcal{O}(V^2)$
BFS	Single-Source Shortest Path	Unweighted	$\mathcal{O}(V+E)$
Dijkstra	Single-Source Shortest Path	Non-Negative Weighted	$\mathcal{O}(E \log V)$ or $\mathcal{O}(V^2)$
Bellman-Ford	Single-Source Shortest Path	Negative Weighted; Non-Negative Cycle	$\mathcal{O}(VE)$
Floyd-Warshall	All-Pair Shortest Path	Negative Weighted; Non-Negative Cycle	$\mathcal{O}(V^3)$

Sample Problems

True or False?

- For a search starting at node *s* in graph *G*, the DFS tree is never the same as the BFS tree.
- If a connected undirected graph G has the same weights for every edge, then every spanning tree of *G* is a minimum spanning tree.
- If a weighted undirected graph has two MSTs, then its vertex set can be partitioned into two, such that the minimum weight edge crossing the partition is not unique.
- If the vertex set of a weighted undirected graph can be partitioned into two, such that the minimum weight edge crossing the partition is not unique, then the graph has at least two MSTs.

True or False? (cont’d)

- Implementations of Dijkstra’s and Kruskal’s algorithms are identical except for the relaxation steps.

- A DFS tree is a spanning tree.

Internet Routing

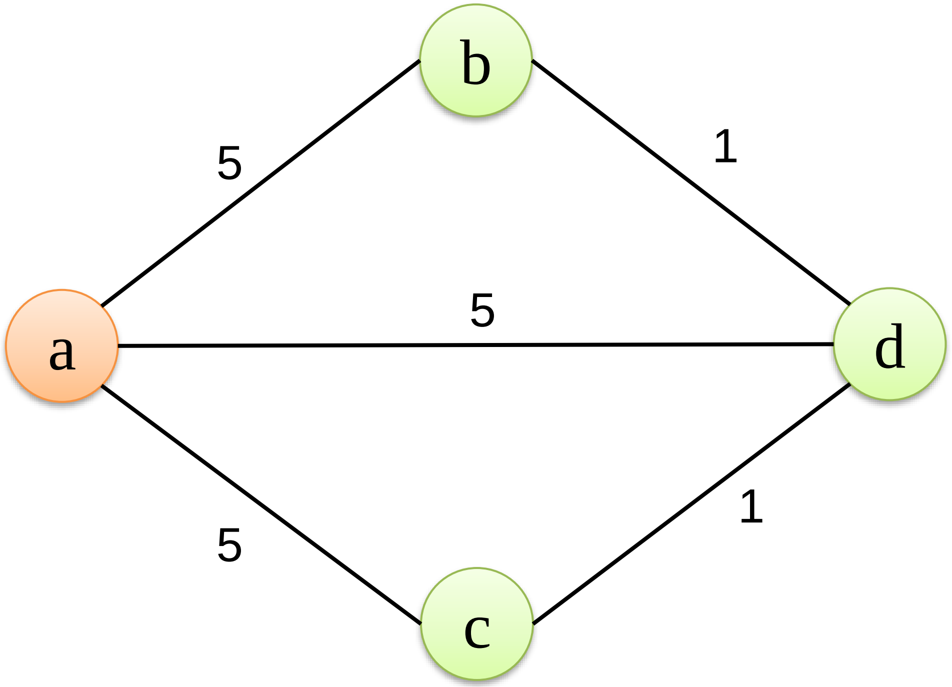
- In Internet routing, there are delays on lines but also, more significantly, delays at routers. Suppose that in addition to having edge lengths $\{l_e: e \in E\}$, a graph also has vertex costs $\{c_v: v \in V\}$.
- Now define the cost of a path to be the sum of its edge lengths, plus the costs of all vertices on the path (including the endpoints). Give an efficient algorithm for the following problem.
  - **Input:** A directed graph $G = (V; E)$; positive edge lengths l_e and positive vertex costs c_v; a starting vertex $s \in V$.
  - **Output:** An array cost such that for every vertex u, cost[u] is the least cost of any path from s to u (i.e., the cost of the cheapest path), under the definition above. Notice that cost[s] = c_s.

- Show that for a graph with distinct edge weights, there is a unique MST.

- Prove or disprove the following:
  - The shortest path between any two nodes in the minimum spanning tree $T = (V, E')$ of connected weighted undirected graph $G = (V, E)$ is a shortest path between the same two nodes in G. Assume the weights of all edges in G are unique and larger than zero.

Does Dijkstra's algorithm give MST?

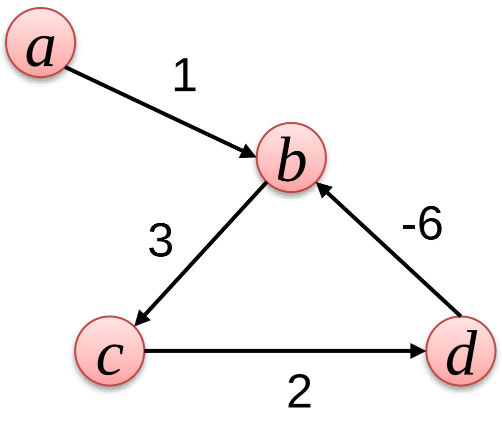
- Find Prim's and Dijkstra's solutions. Assume *a* is the source vertex.
- Dijkstra's algorithm does NOT give MST.



- Often there are multiple shortest paths between two nodes of a graph. Modify Dijkstra’s algorithm so that it computes the shortest path and tracks the number of distinct shortest paths from a start node *s* to all nodes, on a graph with positive weights.

Negative Cycles

- Negative cycle: a cycle whose edges sum to a negative value.
  - There is no shortest path between any pair of vertices i, j which form part of a negative cycle
  - Because path-lengths from i to j can be arbitrarily small (negative).



  - How do *Bellman-Ford* and *Floyd-Warshall* algorithms behave when the input graph has a negative cycle?
    - They can detect it!

Floyd-Warshall Algorithm: Detecting a Negative Cycle (1)

- We want the algorithm to return *null* when a cycle is detected.
- How does the final matrix look like for a graph with a negative cycle after running Floyd-Warshall algorithm?

```
Floyd-Warshall(W) {
    // W is an adjacency matrix
    n = W.rows
    for i = 1 to n
        for j = 1 to n
            A[i,j] = w[i,j]

    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                A[i,j] = min(A[i,j], A[i,k] + A[k,j])
    return A
}
```

Floyd-Warshall Algorithm: Detecting a Negative Cycle (2)

- The algorithm iteratively revises path lengths between all pairs of vertices (i , j), including where $i = j$.
- Initially, the length of the path (i, i) is zero.
- A path $\{i , k, \dots , i\}$ can only improve upon this if it has length less than zero, i.e., denotes a negative cycle.
- Thus, after the algorithm, (i, i) will be negative if there exists a negative-length path from i back to i.

Floyd-Warshall Algorithm: Detecting a Negative Cycle (3)

```
Floyd-Warshall(W) {
    // W is an adjacency matrix
    n = W.rows
    for i = 1 to n
        for j = 1 to n
            A[i,j] = w[i,j]

    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                A[i,j] = min(A[i,j], A[i,k] + A[k,j])

    for i = 1 to n
        if A[i,i] < 0
            return null

    return A[:,:]
}
```

Constructing a Shortest Path (1)

- How do you modify Floyd-Warshall algorithm to print the shortest path between any given vertex i and vertex j?
  - Predecessor matrix $\Pi = (\Pi[i,j])$ stores the predecessor of vertex j on some shortest path from vertex i. If $i = j$ or such path doesn't exist, it's equal to null.
  - Given Π, one can print the shortest path between two vertices as follows.

```
Print-All-Pairs-Shortest-Path(Π, i, j) {
    if i = j
        print i
    else if Π[i, j] = null
        print “no path from” + i + “to” + j + “exists”
    else
        Print-All-Pairs-Shortest-Path(Π, i, Π[i, j])
        print j
}
```

How can you find Π for a given matrix?

Constructing a Shortest Path (2)

- We can calculate Π recursively.
- Initially, we have:

$$\Pi[i,j,0] = \begin{cases} null & \text{if } i = j \text{ or } w[i,j] = \infty \\ i & \text{otherwise} \end{cases}$$

- The recursive equation can be written as

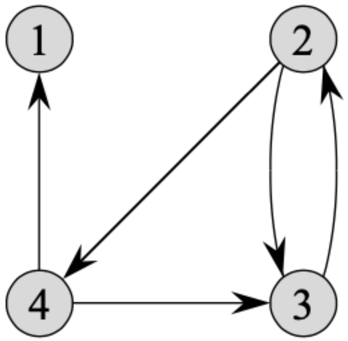
$$\Pi[i,j,k] = \begin{cases} \Pi[i,j,k-1] & \text{if } A[i,j,k-1] \leq A[i,k,k-1] + A[k,j,k-1] \\ \Pi[k,j,k-1] & \text{otherwise} \end{cases}$$

Constructing a Shortest Path (3)

```
Floyd-Warshall(W) {
    // W is an adjacency matrix
    n = W.rows
    for i = 1 to n
        for j = 1 to n
            A[i,j] = w[i,j]
            if i = j OR w[i, j] = ∞
                Π[i,j] = null
            else
                Π[i,j] = i

    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                if A[i,j] > A[i,k] + A[k,j]
                    A[i,j] = A[i,k] + A[k,j]
                    Π[i,j] = Π[k,j]

    return A, Π
}
```

Transitive Closure of a Directed Graph (1)

- Transitive closure of a graph $G=(V, E)$ is defined as $G^*=(V, E^*)$, where $E^*=\{(i, j)$: there is a path between vertex i and vertex $j\}$
- Example:



$$E^* = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

How can you find the transitive closure of a graph?

Transitive Closure of a Directed Graph (2)

- Method 1:
  - Assign weight 1 to every edge of graph G.
  - Run Floyd-Warshall algorithm on the graph and then fill out the adjacency matrix as follows:

$$T[i,j] = \begin{cases} 0 & \text{if } A[i,j] = \infty \\ 1 & otherwise \end{cases}$$

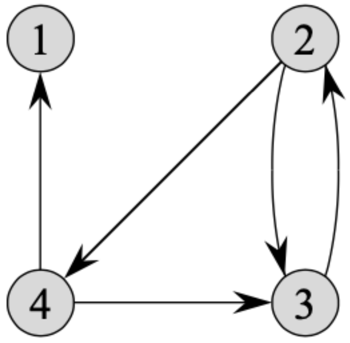
  - Runtime complexity: $\mathcal{O}(|V|^3)$

Transitive Closure of a Directed Graph (2)

- Method 2:
  - Idea: Instead of calculating the actual distance, only keep the result as a binary value.
  - This saves computation time and space.
  - Runtime complexity: $\mathcal{O}(|V|^3)$

```
Transitive-Closure(W) {
    // W is an adjacency matrix
    n = W.rows
    // T is an n-by-n matrix
    for i = 1 to n
        for j = 1 to n
            if w[i,j] > 0
                T[i,j] = 1
            else
                T[i,j] = 0
    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                T[i,j] = T[i,j] OR (T[i,k] AND T[k,j])
    return T
}
```

Transitive Closure of a Directed Graph (3)



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$