CSCI 340 Data Structures and Algorithm Analysis

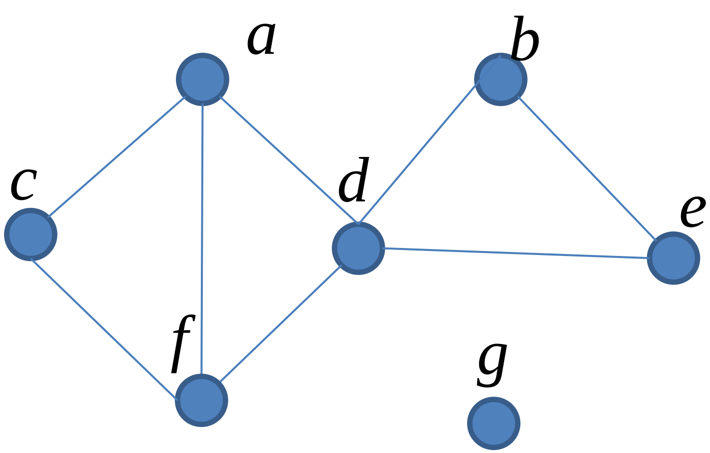
Graph Part II

Graph traversal/search

- Visit all vertices once and only once
- Different from tree traversal:
  - There may be cycles, which can cause infinite loops
  - There may be isolated vertices or isolated sub-graphs

Graph traversal/search

- Breadth-first-traversal
  - Visit siblings first before visiting children.



$a \rightarrow c \rightarrow d \rightarrow f$
$b \rightarrow d \rightarrow e$
$c \rightarrow a \rightarrow f$
$d \rightarrow a \rightarrow b \rightarrow e \rightarrow f$
$e \rightarrow b \rightarrow d$
$f \rightarrow a \rightarrow c \rightarrow d$
g

a, c, d, f, b, e, g
(a,c), (a,d), (a,f), (d,b), (d,e)

Breadth-first-search

- Pseudo-code

```
For all vertices v, num(v) ← 0
S ← ∅   // S records edges
i ← 1
While there is a vertex v s.t. num(v) == 0
    num(v) ← i++
    Q.enqueue ( v )
    while Q is not empty
        v ← Q.dequeue
        for all vertices u adjacent to v
            if  num(u) is 0
                num(u) ← i++
                Q.enqueue(u)
                S ← S + (v,u)
```

S: a sequence of edges visiting the vertices

num(v): records the order of visited vertices

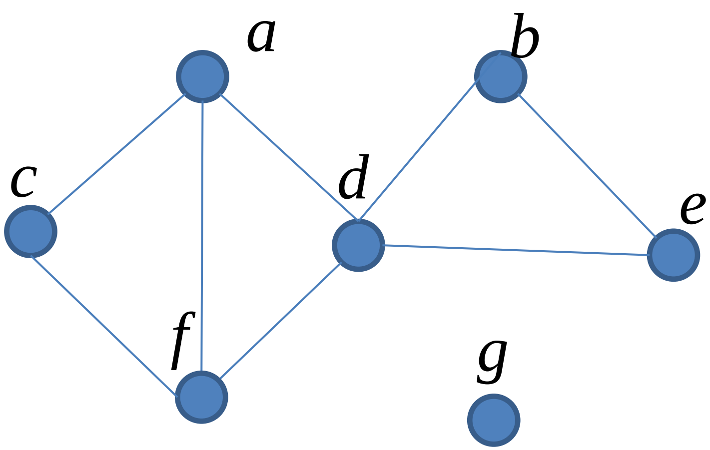
Time efficiency:

Using adjacency list: $O(|V|+|E|)$

Using adjacency matrix: $O(|V|^2)$

Graph traversal/search

- Depth-first-traversal
  - Visit children before siblings
  - Recursive



$a \to c \to d \to f$
$b \to d \to e$
$c \to a \to f$
$d \to a \to b \to e \to f$
$e \to b \to d$
$f \to a \to c \to d$
g

a, c, f, d, b, e, g
(a,c), (c,f), (f, d), (d, b), (b, e)

Depth-first-search

DepthFirstSearch

For all vertices v, num(v) ← 0

S ← Ø // S records edges

i ← 1

While there is a vertex v s.t. num(v) == 0

DFS(v)

S: a sequence of edges visiting the vertices

num(v): records the order of visited vertices

Time efficiency:

Using adjacency list: $O(|V|+|E|)$

Using adjacency matrix: $O(|V|^2)$

DFS(v)

num(v) ← i++

for all vertices u adjacent to v

if num(u) is 0

S ← S + (v,u)

DFS(u)

Shortest path problems

- Weighted graph
  - A numeric value assigned to every edge
- Single pair shortest path problem
  - Finding a path between two vertices that the sum of weights of its edges is minimal
  - If weights are driving distances → shortest distance
  - If weights are driving time → quickest way
- Single source shortest path problem
  - Finding shortest paths from one vertices to all others
- All pair shortest path problem
  - Finding shortest paths for all pairs of vertices

Single pair and single source shortest path problems

- Dijkstra's algorithm
  - Assumptions
    - Graph does not have non-negative weights
    - The weight between a pair of vertices with no edge is infinite

Steps:

1. Find the shortest path from the source to its nearest vertex
2. Find the shortest path from the source to its 2nd nearest vertex
3. Find the shortest path from the source to its 3rd nearest vertex

…

i-1: Find the shortest path from the source to its (i-1)th nearest vertex

Dijkstra's algorithm (cont.)

At step i:

- T_i – the source, the vertices, and edges form a tree
- Fringe vertices – the set of vertices adjacent to vertices in T_i
  - Since weights are non-negative, these vertices are candidates of the i^{th} nearest vertex to the source

For each fringe vertex u (which is adjacent to T_i)

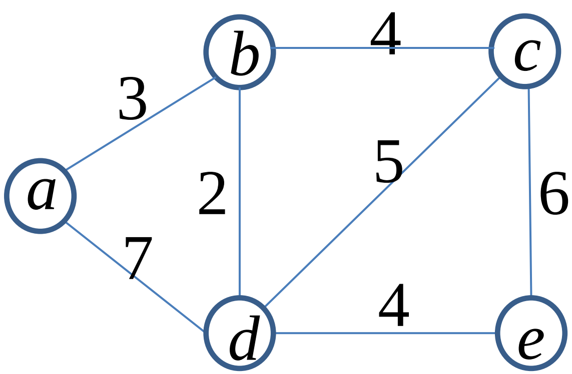
Find its nearest vertex $v \in T_i$

$d_v \leftarrow$ distance between v and source

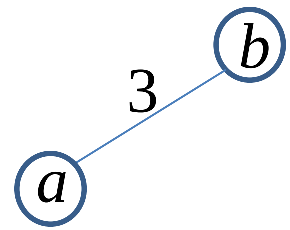
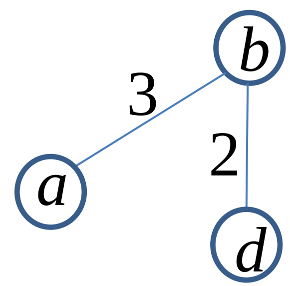
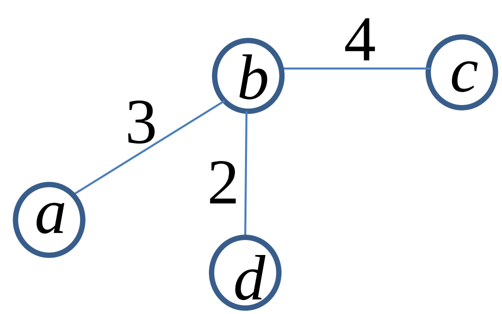
$d_u \leftarrow \min(d_v + \text{weight}(u,v)$, previous d_u if it exists)

Find the minimum d_u. u is the (i-1)th nearest vertex to source.

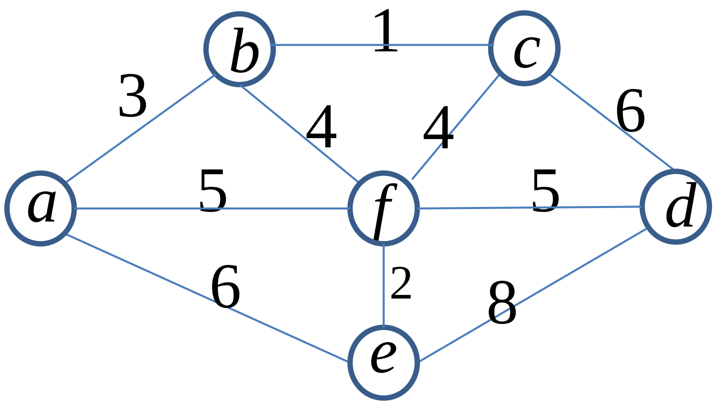
Dijkstra's algorithm example



Source: *a*

	T_i	Fringe vertices
Step 1:		b (a, 3) * c (∞) d (a, 7) e (∞)
Step 2:		c (b, 3+4) d (b, 3+2) * e (∞)
Step 3:		c (b, 3+4) * e (∞)
Step 4:		e (d, 3+2+4)

Another example



Source: a

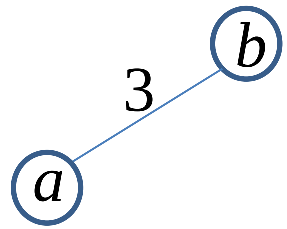
T_i

Fringe vertices

Step 1:

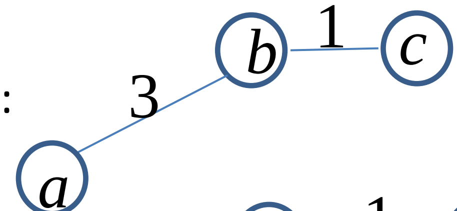
b $(a, 3)$ *
c (∞)
d (∞)
e $(a, 6)$
f $(a, 5)$

Step 2:



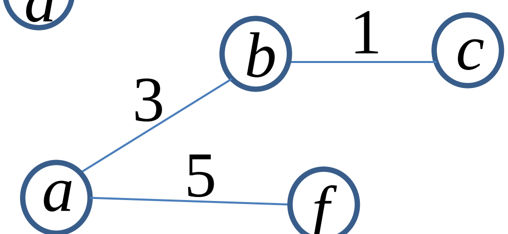
c $(b, 3+1)$*
d (∞)
e $(a, 6)$
f $(a, 5)$

Step 3:



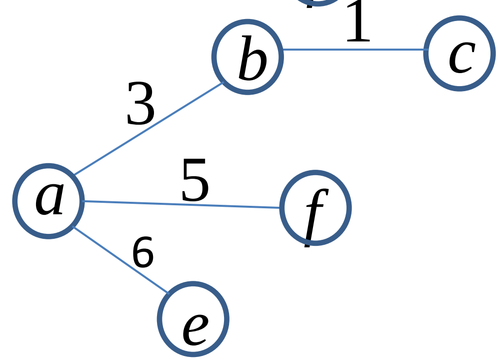
d $(c, 10)$
e $(a, 6)$
f $(a, 5)$ *

Step 4:



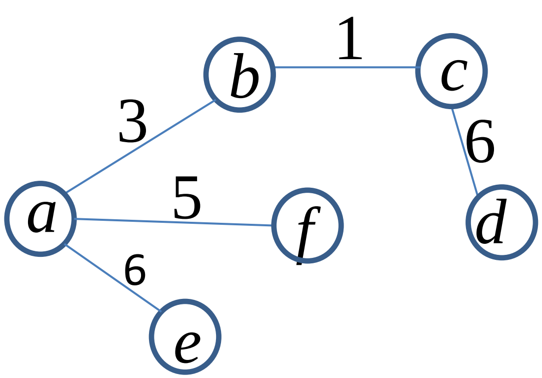
d $(c, 10)$
e $(a, 6)$ *

Step 4:

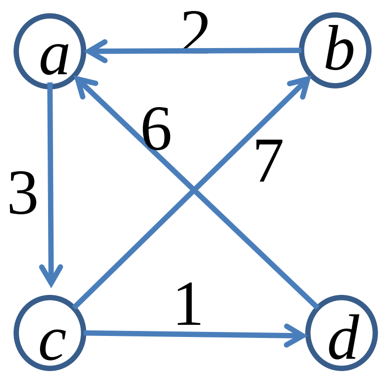


d $(c, 10)$*

Step 5:



All pairs shortest path problem



Weight matrix

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Distance matrix

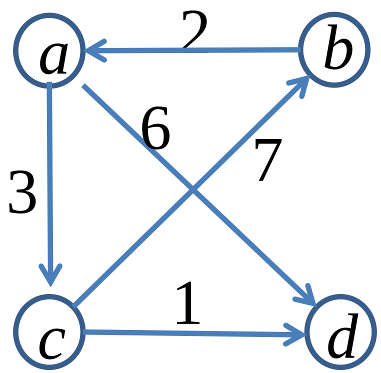
	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

All pairs shortest path problem

- Floyd-Warshal algorithm
  - A graph with n vertices, each labeled with 1, 2, … n.
  - Weighted directed or undirected graph
  - No negative cycle
  - Compute through a series of $n \times n$ matrices

$$D^{(0)}, \ldots, D^{(k)}, \ldots, D^{(n)}$$

$D^{(0)}$ is the weight matrix
$D^{(k)}$ is contains shortest paths with intermediate vertices of number at most k
$D^{(n)}$ is the final distance matrix containing shortest paths



$D^{(0)}$: weight matrix

	a	b	c	d
a	0	∞	3	6
b	2	0	∞	∞
c	∞	7	0	1
d	∞	∞	∞	0

$D^{(1)}$: only use a as intermediate vertex

	a	b	c	d
a	0	∞	3	6
b	2	0	5	8
c	∞	7	0	1
d	∞	∞	∞	0

$D^{(2)}$: use a and b as intermediate vertices

	a	b	c	d
a	0	∞	3	6
b	2	0	5	8
c	9	7	0	1
d	∞	∞	∞	0

$D^{(3)}$: use a, b, c as intermediate vertices

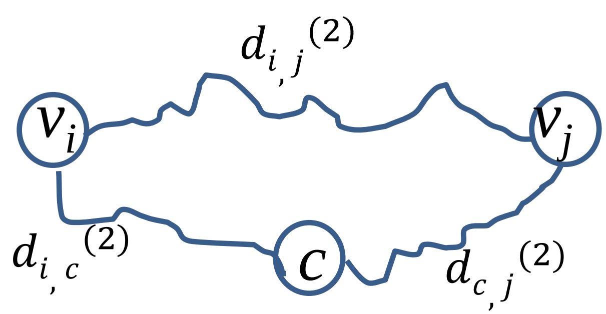
	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	∞	∞	∞	0

$D^{(4)}$: use a, b, c, d as intermediate vertices

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	∞	∞	∞	0

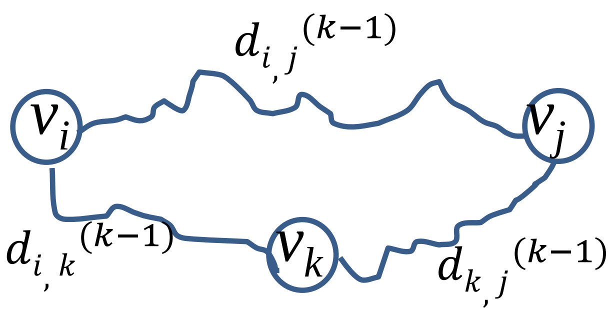
Floyd-Warshall algorithm

- Suppose we have the shortest paths allowing using at most a and b as intermediate vertices
  - Now if we allow c as intermediate vertex, what will be the shortest path between any pair of vertices (e.g. $v_i \rightarrow v_j$)?
  - Either use c or not to use c



Floyd-Warshall algorithm

- Suppose we have the shortest paths allowing using intermediate vertices no larger than k-1
  - If we allow kth vertex as intermediate vertex, what will be the shortest path from v_i to v_j ?
  - $d_{i,j}^{(k)}$ = min($d_{i,j}^{(k-1)}$, $d_{i,k}^{(k-1)}$+ $d_{k,j}^{(k-1)}$) for $k \geq 1$
  - $d_{i,j}^{(0)}$ = $w_{i,j}$



Floyd-Warshal algorithm

$D \leftarrow$ weight matrix

For k from 1 to n

    for i from 1 to n

        for j from 1 to n

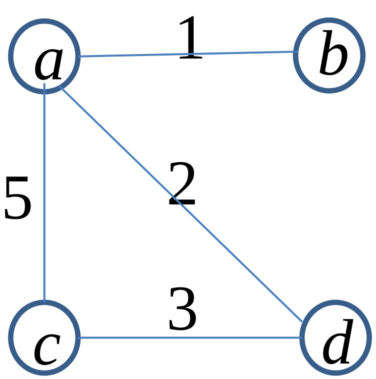
            $D[i, j] \leftarrow \min (D[i, j], D[i, k] + D[k, j])$

Minimum Spanning Tree

- Acyclic graph: a graph that does not contain a cycle
- Connected graph: there is a path from any vertex to any other vertex in the graph
- Spanning tree of a connected graph is its connected acyclic subgraph, i.e., a tree, that contains all the vertices of the graph.

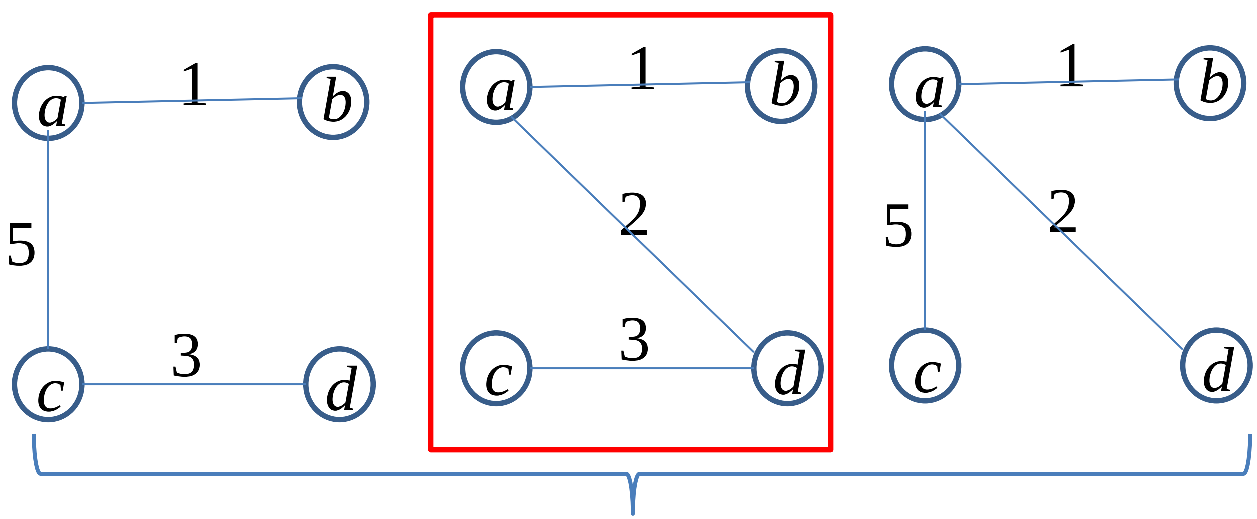
Minimum Spanning Tree

- M.S.T. of a weighted connected graph is its spanning tree of the smallest weight, where the weight of the tree is defined as sum of the weights on all its edges
- Example:



A given graph

M.S.T.



Spanning trees

M.S.T.

- Application example: suppose you have a business with quite many branch offices and you want to lease phone lines to connect them with each other
- Goal: connect all branches with minimum cost
- Algorithms:
  - Prim's algorithm
  - Kruskal's algorithm

M.S.T. – Prim's algorithm

Input: G // $G = <V, E>$

$V_T \leftarrow \{V_0\}$ // V-V_T denotes vertices not in V_T

$E_T \leftarrow \emptyset$ // record current M.S.T.

For i from 1 to $|V|$-1

    Find a minimum weight edge $e^*=(v^*,u^*)$ among all edges (v, u) where $v \in V_T$ **AND** $u \in V$-V_T

    $V_T \leftarrow V_T \cup \{u*\}$

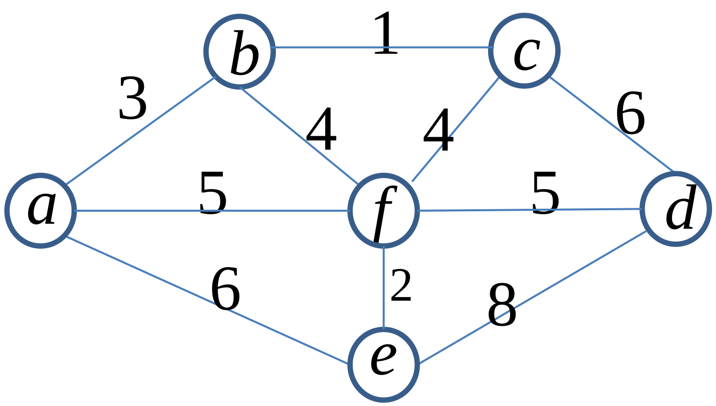
    $E_T \leftarrow E_T \cup \{(v^*,u^*)\}$

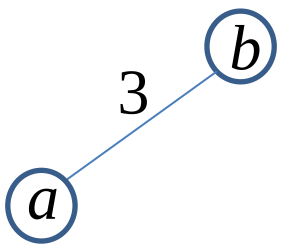
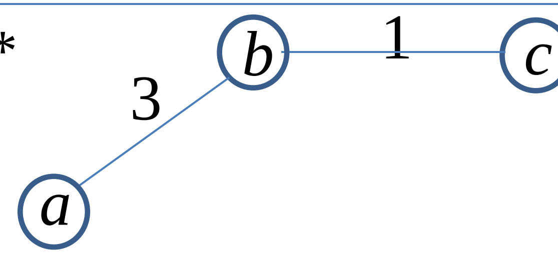
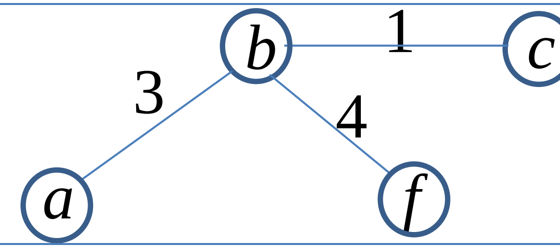
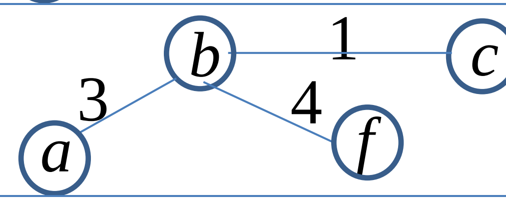
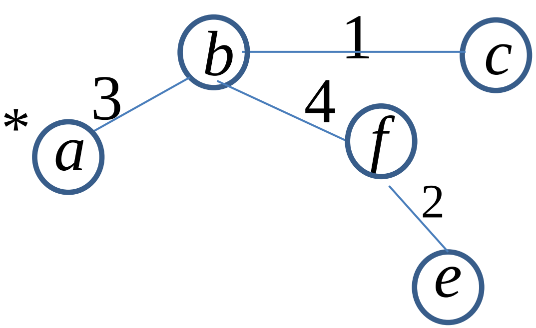
Return E_T

Time cost: O ($|V|^2$)

or O ($|E| \times \lg|V|$) using min-heap

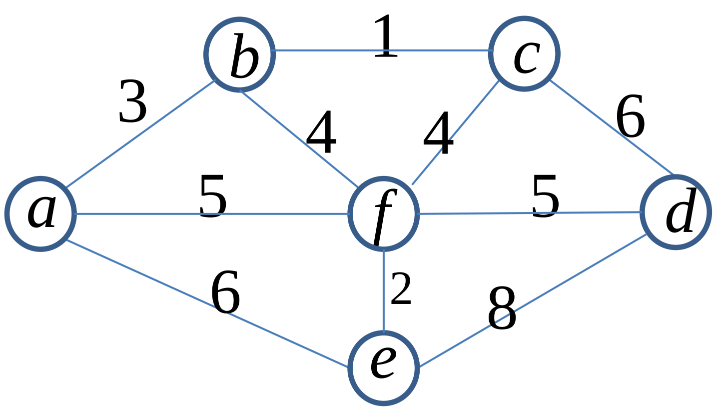
M.S.T. Prim’s algorithm



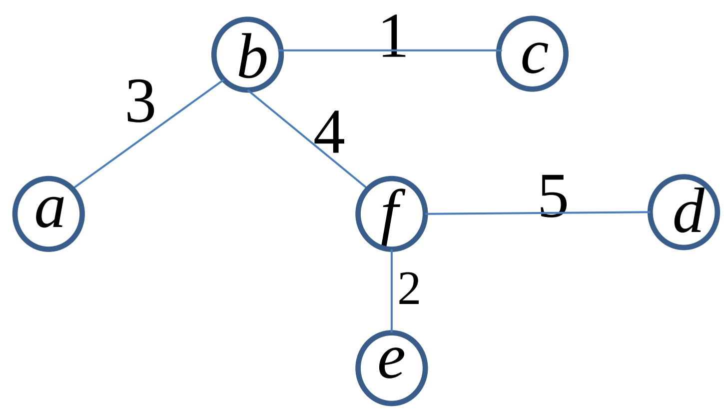
	V_T	V-V_T	T
i=1	a	b (a, 3)* c (-, ∞) d (-, ∞) e (a, 6) f (a, 5)	
i=2	a, b	c (b, 1)* d (-, ∞) e (a, 6) f (b, 4)	
i=3	a,b,c	d (c, 6) e (a, 6) f (b, 4)*	
i=4	a,b,c,f	d (f, 5) e (f, 2)*	
i=5	a,b,c,f,e	d (f, 5)*	
	a,b,c,f,e,d		

M.S.T.

Input graph:



Output MST:



Sum of weight: 15

M.S.T. -- Kruskal's algorithm

Initialize every vertex as a separate tree

$E_T \leftarrow \emptyset$

$S \leftarrow$ sort (E)

While S is not empty

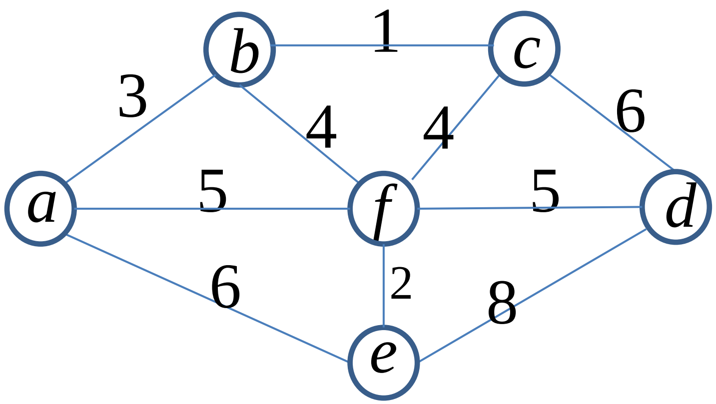
    remove minimum weight edge e from S

    if e connects two different trees

        add e to E_T, combine two trees into one

Time cost: O($E \times \lg V$)

Kruskal’s algorithm

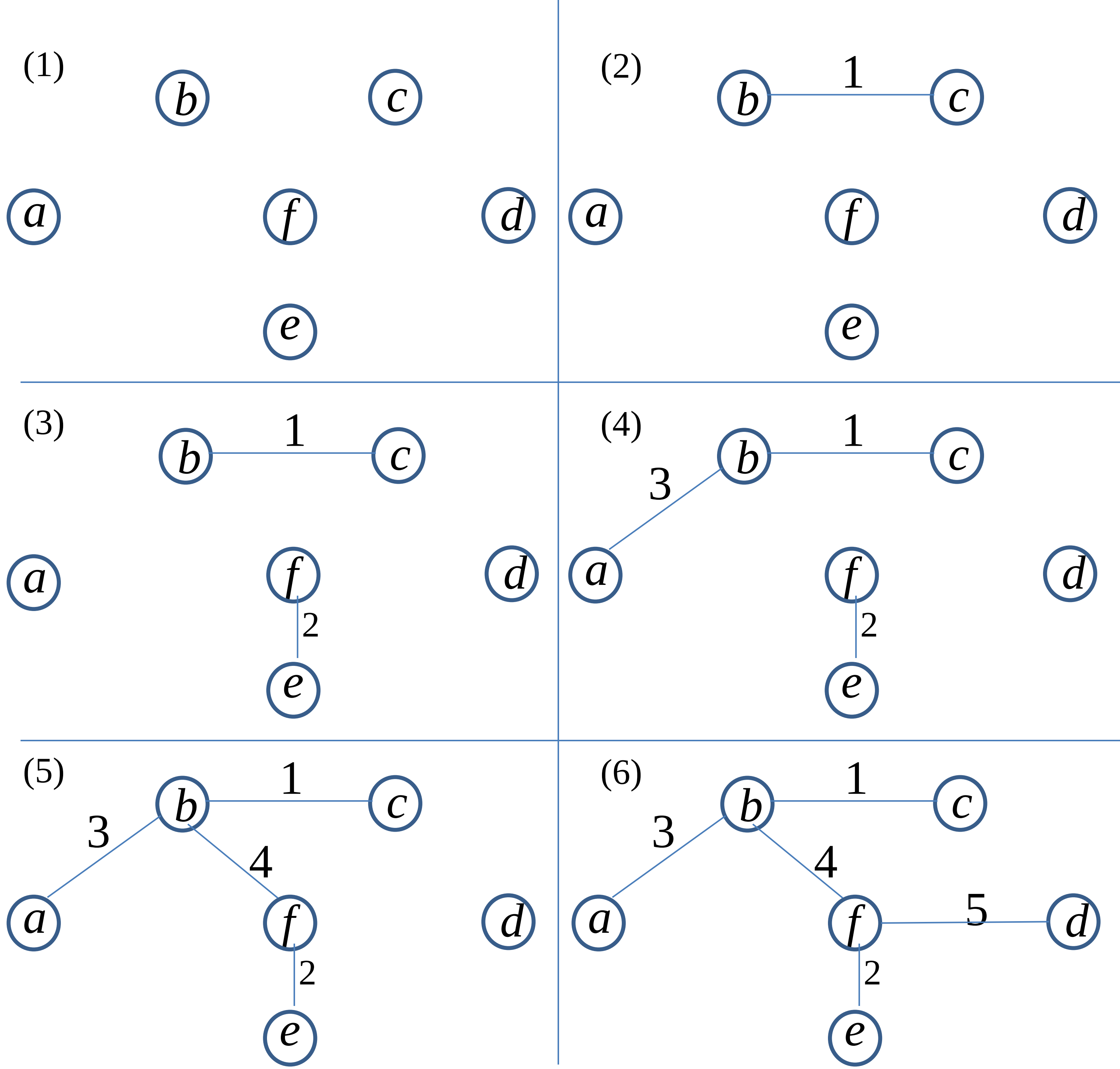


S:

Order	Weight	Edge
1	1	$b \leftrightarrow s$
2	2	$e \leftrightarrow f$
3	3	$a \leftrightarrow b$
4	4	$b \leftrightarrow f$
5	4	$c \leftrightarrow f$
6	5	$d \leftrightarrow f$
7	5	$a \leftrightarrow f$
8	6	$a \leftrightarrow c$
9	6	$c \leftrightarrow d$
10	8	$d \leftrightarrow e$

S:

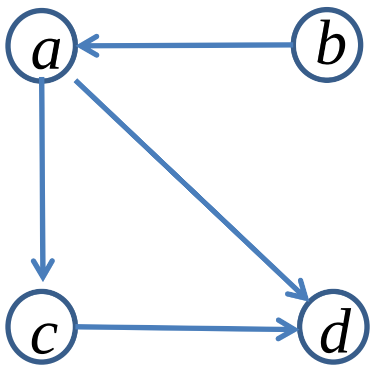
W	Edge
1	$b \leftarrow\rightarrow s$
2	$e \leftarrow\rightarrow f$
3	$a \leftarrow\rightarrow b$
4	$b \leftarrow\rightarrow f$
4	$c \leftarrow\rightarrow f$
5	$d \leftarrow\rightarrow f$
5	$a \leftarrow\rightarrow f$
6	$a \leftarrow\rightarrow c$
6	$c \leftarrow\rightarrow d$
8	$d \leftarrow\rightarrow e$

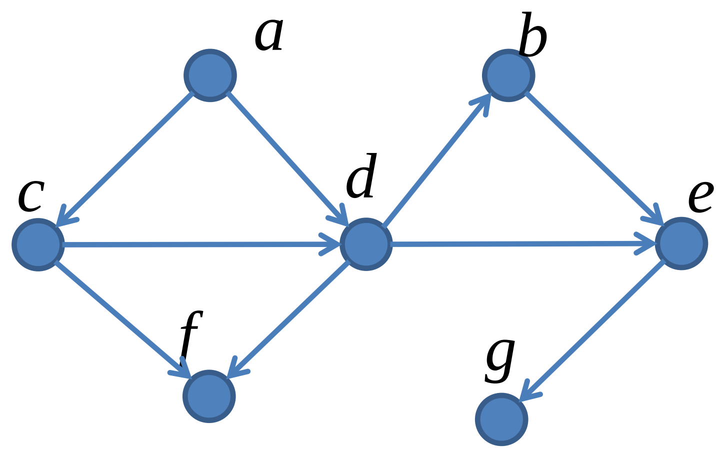


Topological Sort

- Applies to digraph
- The digraph cannot have cycle – otherwise there's no valid topological sort
- Topological sort labels all vertices with numbers 1, 2, ..., $|V|$ so that $i < j$ only if there is a path from vertex v_i to v_j
- Examples

Topological Sort





$b \to a \to c \to d$

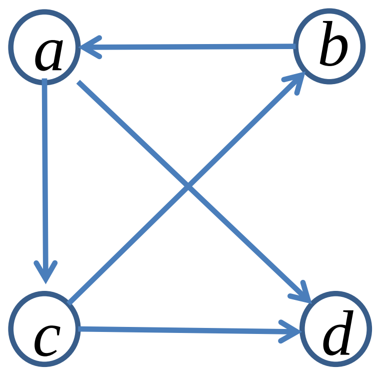
1 2 3 4

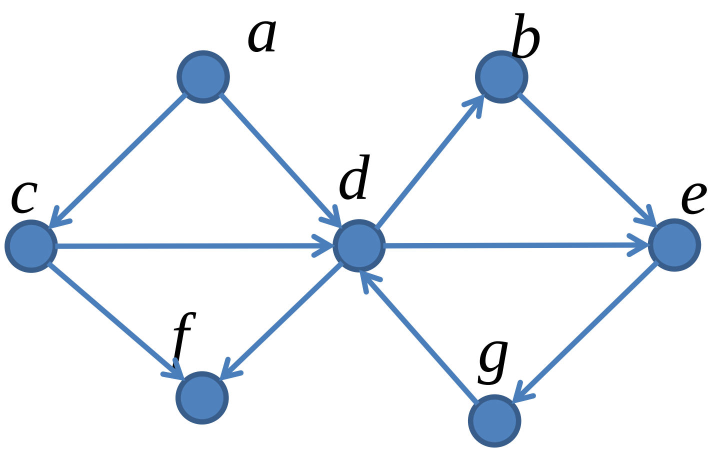
$a \to c \to d \to f \quad b \to e \to g$

1 2 3 4 5 6 7

Topological Sort

- Topological sorting is impossible if the graph contains cycle(s).





Topological Sort – method 1

topologicalSort1(digraph G)

    for all vertices v, num(v) ← 0

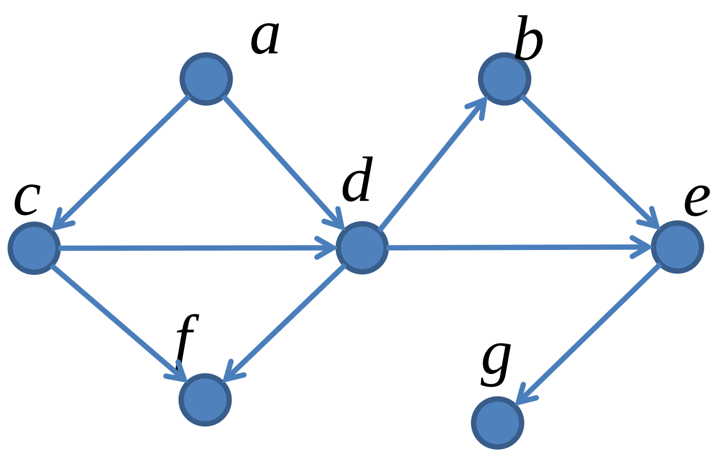
    for i from 1 to $|V|$

        find a sink v // there must be a sink in DAG

        num(v) ← i

        remove v from G and all edges incident with v

    output vertices according to their reverse order in num

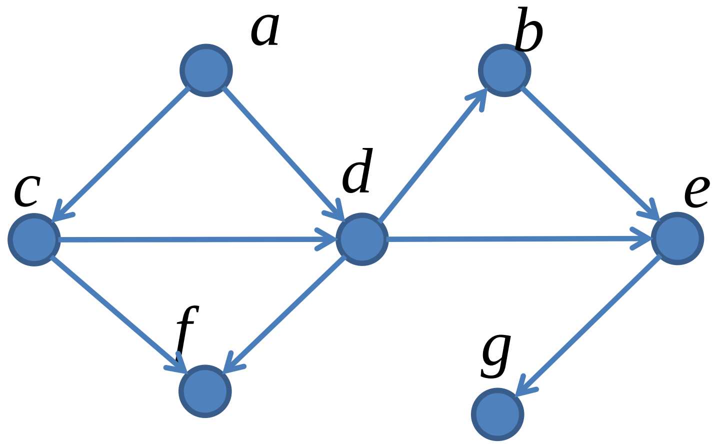


Topological Sort – method 2

```
TS ( v )
    num(v) ← i++
    for all vertices u adjacent to v
        if num(u) is 0
            TS(u)
        else
            if TSNum(u) is 0
                error // indicate cycle
    TSNum(v) ← j++
```

```
topologicalSort2 (digraph G)
    for all vertices v
        num(v)  ← TSNum(v) ← 0
        i ← j ← 1
    while there is a vertex v s.t.
                          num(v) is 0
        TS(v)
    output vertices according to
            reverse order in TSNum
```

Topological Sort



- Start with vertex *a*

	num	TSNum
a	1	7
b	4	3
c	2	6
d	3	5
e	5	2
f	7	4
g	6	1

- Start with vertex *c*

	num	TSNum
a	7	7
b	3	3
c	1	6
d	2	5
e	4	2
f	6	4
g	5	1

Cycle detection

cycleDetectionDFS(v)
    num(v) $\leftarrow$ i++
    for all vertices u adjacent to v
        if num(u) is 0
            $S \leftarrow S \cup$ edge(v,u)
            cycleDetectionDFS(u)
        else // u is visited
            if edge(u,v) $\notin S$
                cycle detected

back edge: a new edge connecting to a vertex already visited.

DepthFirstCycleDetection(simple graph G)
    For all vertices v, num(v) $\leftarrow$ 0
    $S \leftarrow \emptyset$
    $i \leftarrow 1$
    While there is a vertex v s.t. num(v) is 0
        cycleDetectionDFS(v)

Cycle Detection

```
digraphCDetectionDFS(v)
    num(v) ← i++
    for all vertices u adjacent to v
        if num(u) is 0
            digraphCDetectionDFS(u)
        else // u is visited
            if num(u) is not ∞
                cycle detected
    num(v) ← ∞ // Assign a very large
               // value when all v's
               // descendants are
               // visited already
```

```
DepthFirstCycleDetection( digraph G )
    for all vertices v, num(v) ← 0
    i← 1
    while there is a vertex v s.t. num(v) is 0
        digraphCDetectionDFS(v)
```