Data Structures and Algorithms

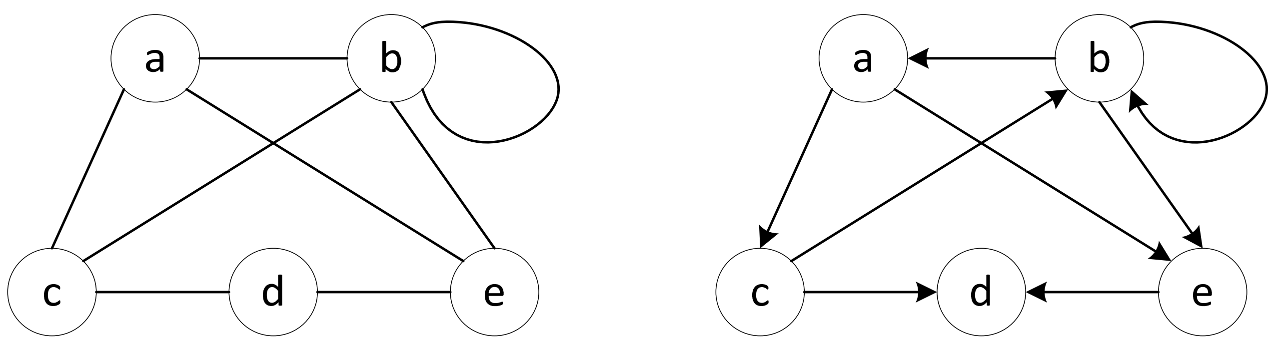
Chapter 14

Graph Algorithms
Basics

- A *graph* is a set *V* of *vertices* and a collection *E* of *edges, G* = (*V*, *E*)
- An edge connecting vertices (or nodes) *u* and *v* is denoted (*u*, *v*).
- An edge can be *directed* or *undirected*.
- Directed graph vs. undirected graph:

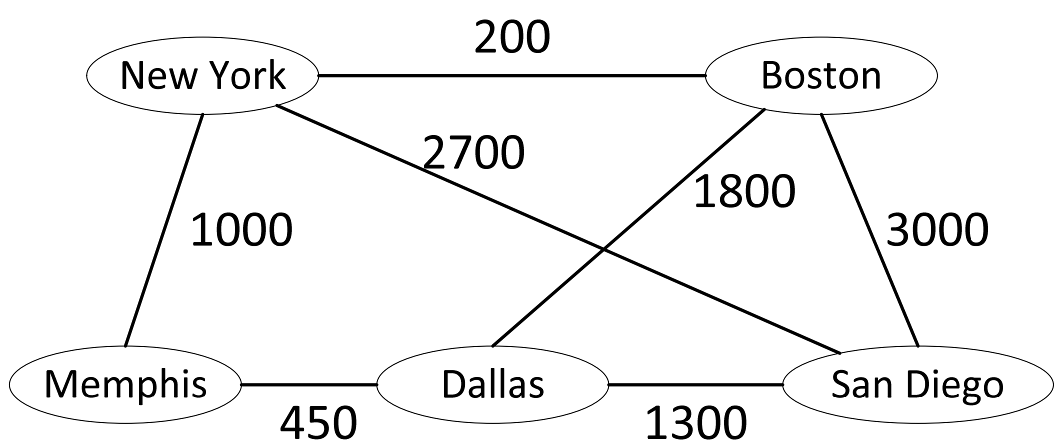
(a) Undirected graph

(b) Directed graph



Graph Algorithms
Basics

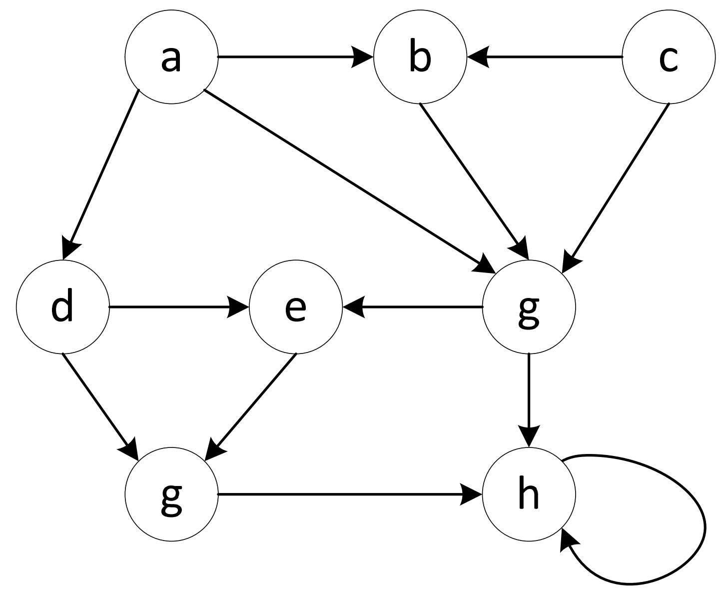
- Two vertices *u* and *v* are said to be *adjacent* if there is an edge (*u*, *v*).
- An edge is said to be *incident* to a vertex if the vertex is one of the edge's endpoints.
- *Weighted* graph: An information (usually called weight) is associated with edges



Graph Algorithms
Basics

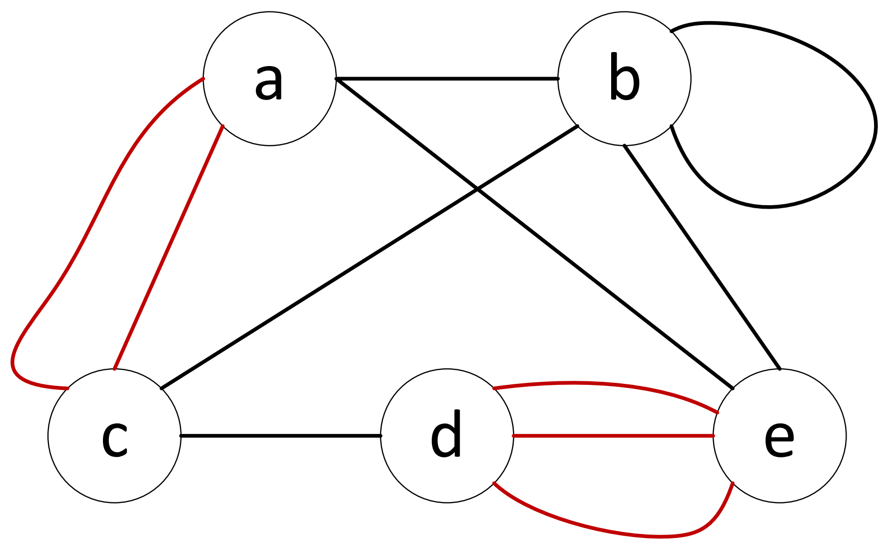
- Outgoing edge vs. incoming edge
- Degree, in-degree, and out-degree of a node

- The outgoing edges of vertex *g* are (*g*, *e*), (*g*, *h*).
- The incoming edges of vertex *g* are (*a*, *g*), (*b*, *g*), (*c*, *g*).
- The degree of vertex *g*, *deg*(*g*) = 5.
- The in-degree of vertex *g*, *indeg*(*g*) = 3.
- The out-degree of vertex *g*, *outdeg*(*g*) = 2.



Graph Algorithms
Basics

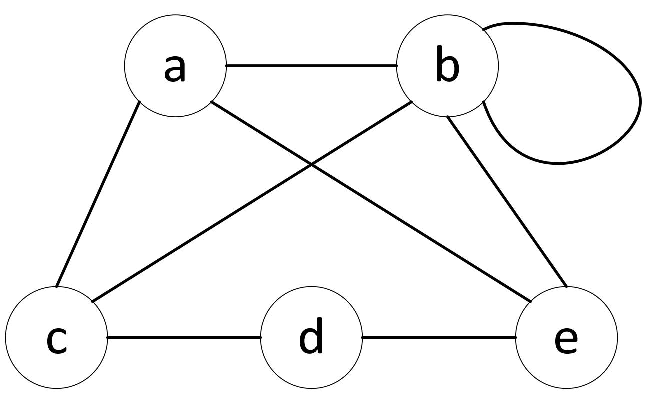
- Parallel edges and self-loops



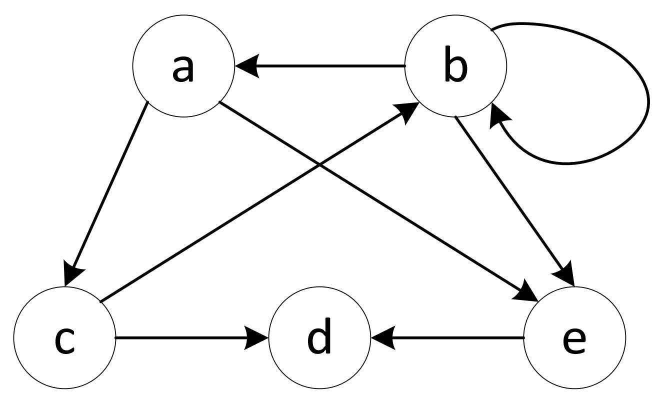
Graph Algorithms
Basics

- *Path*, *cycle*, *simple path*, *simple cycle*, *directed path*, *directed cycle*

(a) Undirected graph



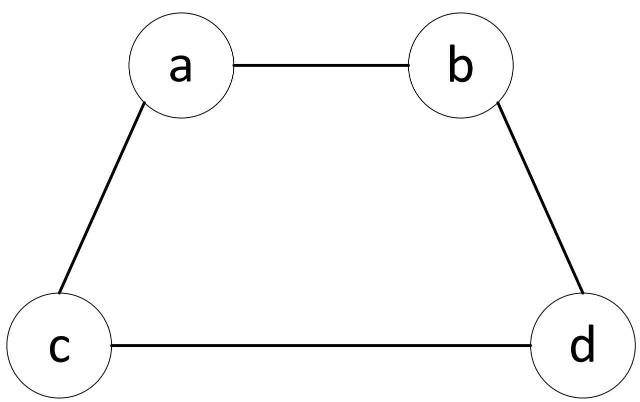
(b) Directed graph



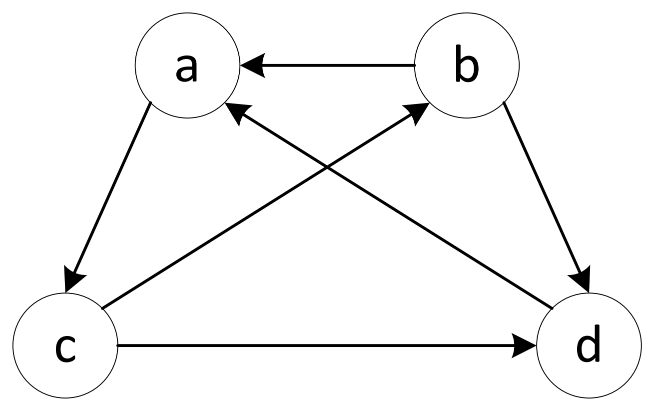
Graph Algorithms
Basics

- *Connected graph* and *strongly connected graph*

(a) Connected graph



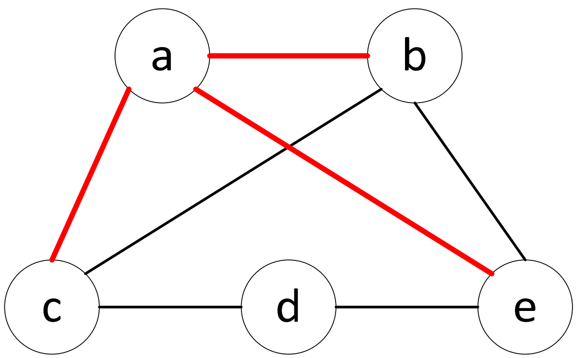
(b) Strongly connected graph



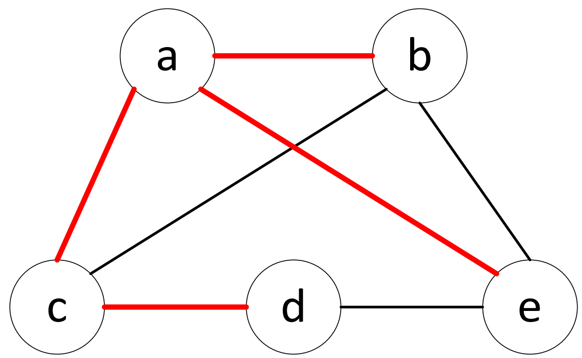
Graph Algorithms
Basics

- *Subgraph* and *spanning subgraph*

(a) A subgraph



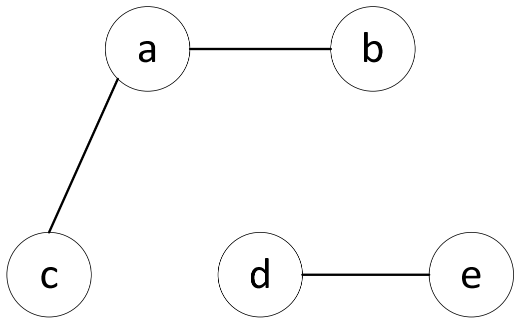
(b) A spanning subgraph



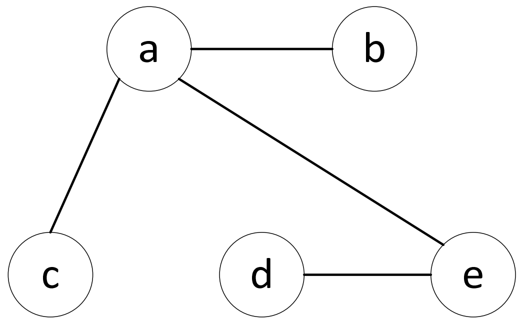
Graph Algorithms
Basics

- *Forest* and *tree*

(a) A forest



(b) A tree



- A *spanning tree* of a graph is a spanning subgraph that is a tree

Graph Algorithms
Basics

- Graph properties
  - If a graph $G = (V, E)$ has m edges, then

    $$\sum_{v \text{ in } V} \deg(v) = 2m$$

  - If $G = (V, E)$ is a directed graph with m edges, then

    $$\sum_{v \text{ in } V} in\deg(v) = \sum_{v \text{ in } V} out\deg(v) = m$$

- Let G be a simple graph with n vertices and m edges.
  - If G is undirected, then $m \le \frac{n(n-1)}{2}$
  - If G is directed, then $m \le n(n - 1)$.

Graph Algorithms
Basics

- Graph properties (continued)
  - Let G be an undirected graph with n vertices and m edges:
    - If G is connected, then $m \geq n - 1$
    - If G is a tree, then $m = n - 1$
    - If G is a forest, then $m \leq n - 1$

Graph Algorithms
Graph ADT

- Oprerations
  - numVertices( )
  - vertices( )
  - numEdges( )
  - edges( )
  - getEdge(u, v)
  - endVertices(e)
  - opposite(v, e)

Graph Algorithms
Graph ADT

- Oprerations (continued)
  - outDegree(*v*)
  - indegree(*v*)
  - outgoingEdges(*v*)
  - incomingEdges(*v*)
  - insertVertex(*x*)
  - insertEdge(*u*, *v*, *x*)
  - removeVertex(*v*)
  - removeEdge(*e*)

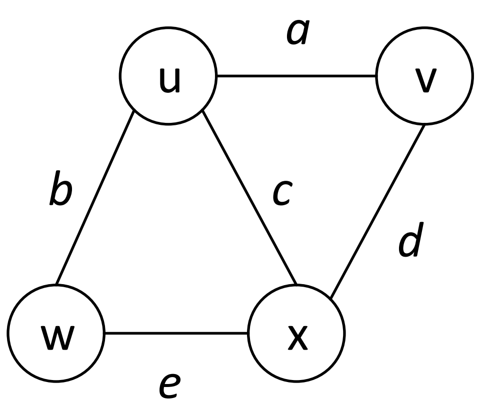
Graph Algorithms
Data Structures for Graphs

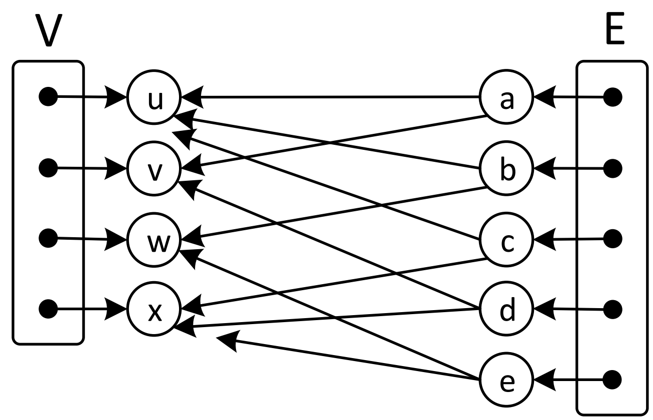
- Edge list, adjacency list, adjacency map, adjacency matrix

Method	**Edge List**	**Adj. List**	**Adj. Map**	**Adj. Matrix**
vertices( )	O(n)	O(n)	O(n)	O(n)
edges( )	O(m)	O(m)	O(m)	O(m)
getEdge(u, v)	O(m)	$O(\min(d_u, d_v))$	O(1) exp.	O(1)
outDegree(v) **inDegree(v)**	O(m)	O(1)	O(1)	O(n)
outgoingEdges(v) **incomingEdges(v)**	O(m)	$O(d_v)$	$O(d_v)$	O(n)
insertVertex(x)	O(1)	O(1)	O(1)	$O(n^2)$
removeVertex(v)	O(m)	$O(d_v)$	$O(d_v)$	$O(n^2)$
insertEdge(u, v, x)	O(1)	O(1)	O(1) exp.	O(1)
remove Edge(e)	O(1)	O(1)	O(1) exp.	O(1)

Graph Algorithms
Data Structures for Graphs

- Edge list

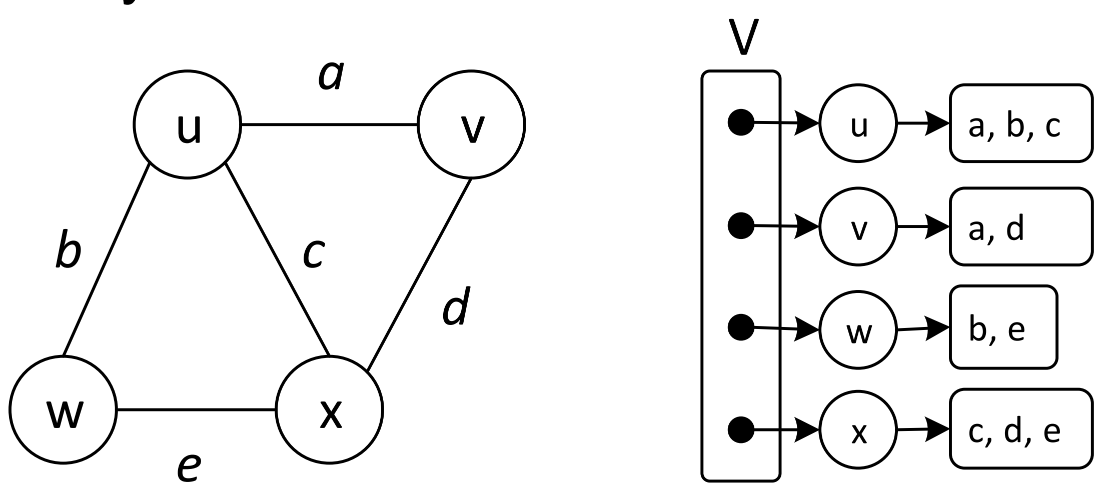




- *V* is a list of vertices and *E* is a list of edges. Both can be implemented using doubly linked lists.

Graph Algorithms
Data Structures for Graphs

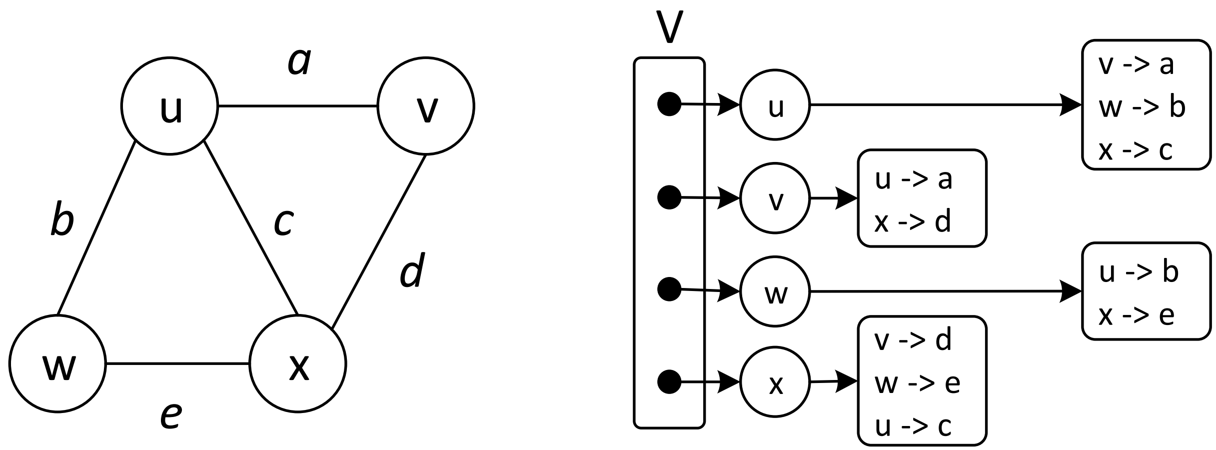
- Adjacency list



- *V* is a list of vertices.
- Each vertex *v* has a reference to a separate collection of edges that are incident to *v*.
- The collection is called *incidence collection*.

Graph Algorithms
Data Structures for Graphs

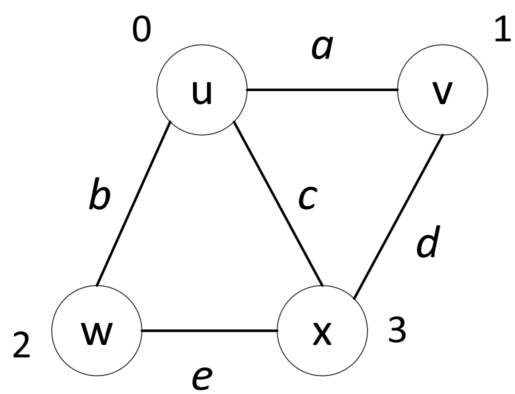
- Adjacency map



- Incidence collection of *V* is implemented as a map.
- Suppose edge *a* = (*u*, *v*) is in the incidence collection of *u*. Then, <*v*, *a*> pair is stored in the map, where *v* is a key and *a* is the corresponding value.

Graph Algorithms
Data Structures for Graphs

- Adjacency matrix



		0	1	2	3
u →	0		a	b	c
v →	1	a			d
w →	2	b			e
x →	3	c	d	e	

- *n* x *n* matrix.
- Vertices are encoded to integers and these integers are used as indexes.
- The entry corresponding to vertices *u* and *v* stores an edge (*u*, *v*).

Graph Algorithms
Graph Traversals

- A *graph traversal* is a systematic procedure for visiting (and processing) all vertices in the graph.
- We say a traversal is efficient if its running time is proportional to the number of vertices and edges in the graph.

- Applications (for directed graph):
  – Find a direct path from vertex *u* to vertex *v*.
  – Find all vertices of *G* that are reachable from a given vertex *s*.
  – Determine whether *G* is acyclic.
  – Determine whether *G* is strongly connected.

Graph Algorithms
Graph Traversals

- Applications (for undirected graph):
  - Find a path from vertex *u* to vertex *v*.
  - Given a start vertex *s*, find a path with the minimum number of edges from *s* to every other vertex.
  - Test whether *G* is connected.
  - Find a spanning tree of *G*.
  - Identify a cycle in *G*.
- Will discuss *depth-first search* (*DFS*) and *bread-first search* (*BFS*).

Graph Algorithms
DFS

- Pseudocode

Algorithm DFS (*G*, *u*)

Input: A graph *G* and a vertrx *u* of *G*

Output: A collection of vertices reachable from *u*, with
their discovery edges

Mark *u* as visited

for each of *u*'s outgoing edges, *e* = (*u*, *v*) do

  if *v* has not been visited then

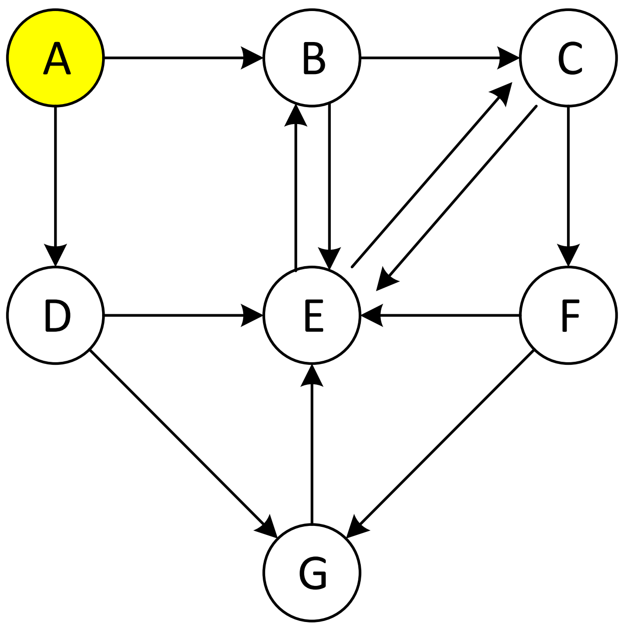
    Record edge e as the discovery edge for vertex *v*

    Recursively call DFS(*G*, *v*)

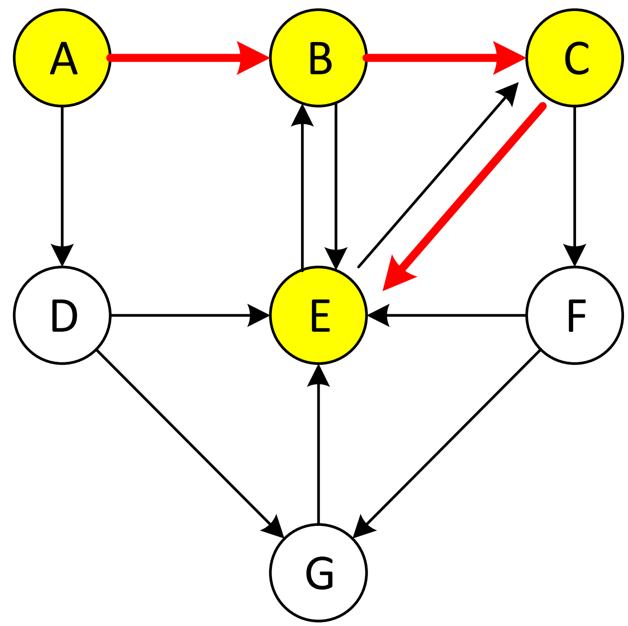
Graph Algorithms
DFS

- Illustration (on a directed graph)

A directed graph
Start at vertex A



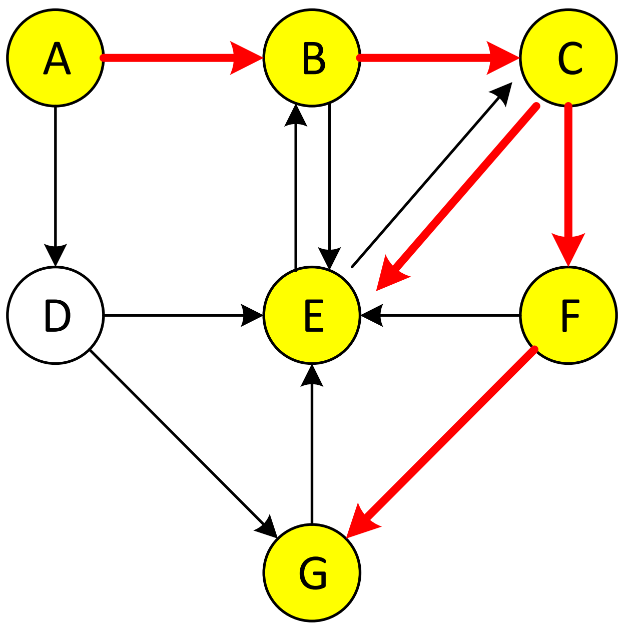
A -> B -> C -> E
Backtrack to C



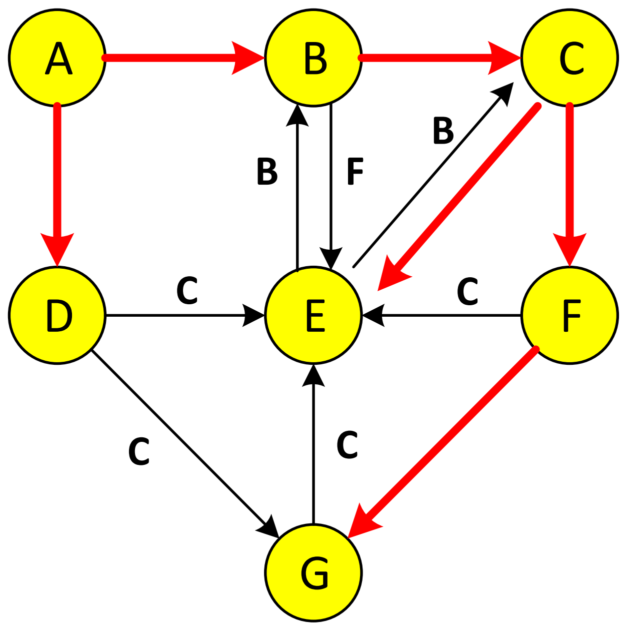
Graph Algorithms
DFS

- Illustration (on a directed graph)

C -> F -> G
Backtrack to F -> C -> B -> A



A -> D
Finished



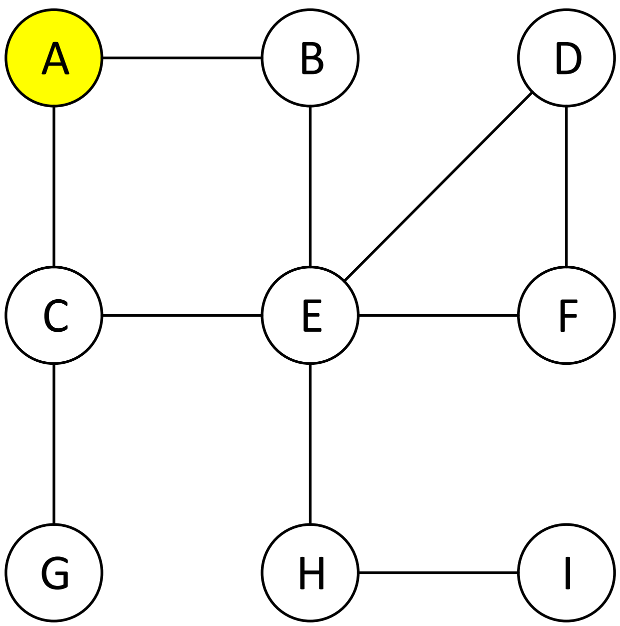
Graph Algorithms
DFS

- Illustration (on a directed graph)
  - Classification of edges:
    - *Back edges*: A back edge connects a vertex to its ancestor in the *DFS* tree. They are labeled *B*.
    - *Forward edges*: A forward edge connects a vertex to its descendant in the *DFS* tree. They are labeled *F*.
    - *Cross edge*: A cross edge connects a vertex to a vertex that is neither its ancestor nor its descendant. They are labeled *C*.

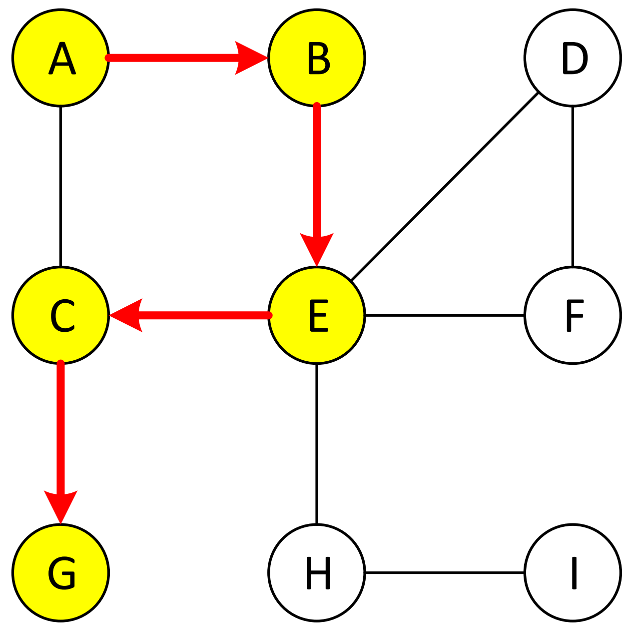
Graph Algorithms
DFS

- Illustration (on an undirected graph)

An undirected graph
Start at vertex A



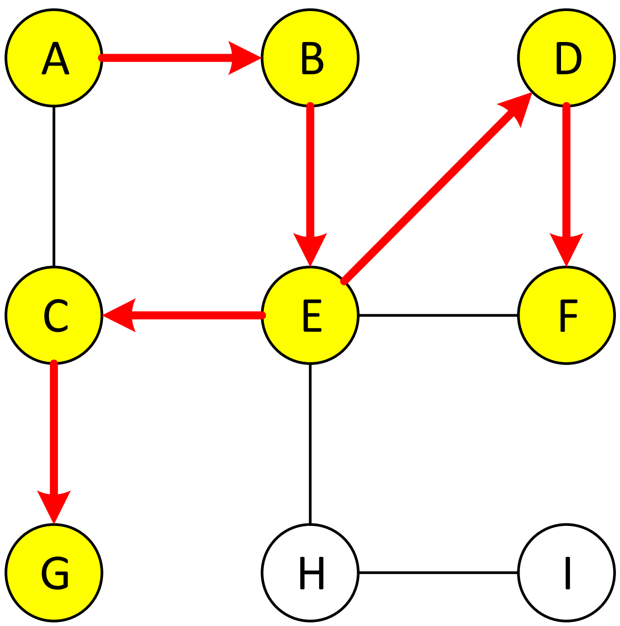
A -> B -> E -> C -> G
Backtrack to C -> E



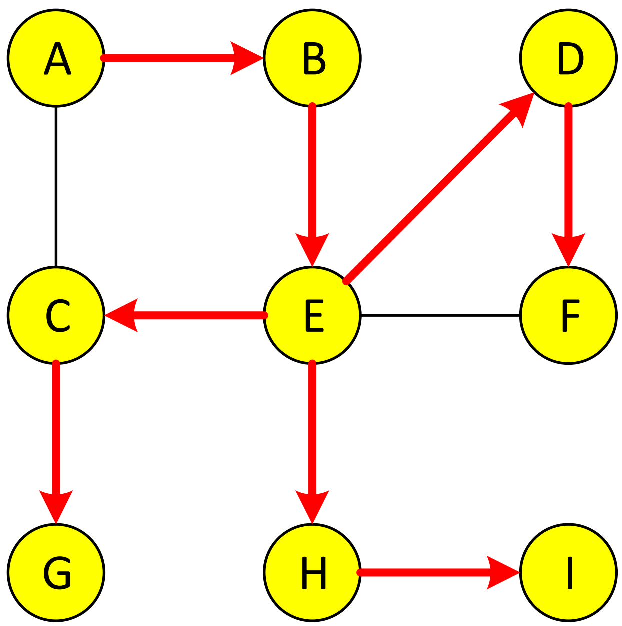
Graph Algorithms
DFS

- Illustration (on an undirected graph)

E -> D -> F
Backtrack to D -> E



E -> H -> I
Finished



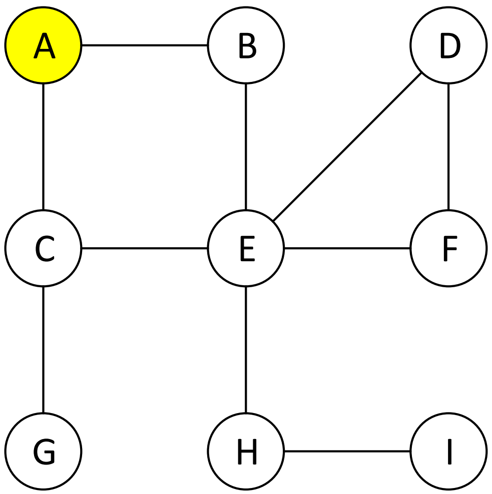
Graph Algorithms
DFS

- DFS properties:
  - A *DFS* on an undirected graph G starting at a vertex *s* visits all vertices in the connected component of *s*, and the discovery edges form a *spanning tree* of the connected component of *s*.
  - A *DFS* on a directed graph G starting at a vertex *s* visits all vertices reachable from *s*, and the *DFS* tree contains the directed paths from *s* to every vertex reachable from *s*.
- Running time: $O(n_s + m_s)$, here n_s is the number of vertices reachable from *s* and m_s is the number of edges that are incident to those vertices

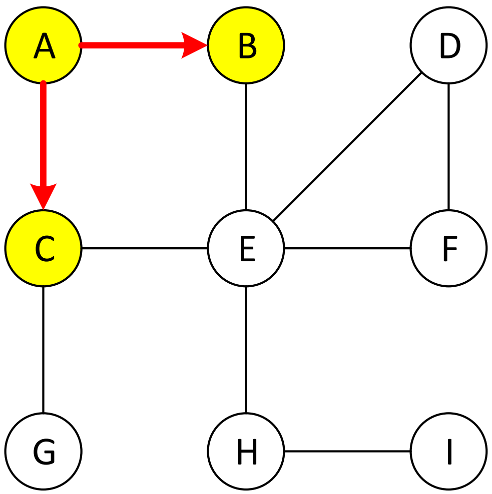
Graph Algorithms
BFS

- Outline
  - Start at the starting vertex *s*
  - Visit all vertices that are "one-edge away" from *s*
  - Visit all vertices that are "two-edge away" from *s*
  - and so on.
- Illustration

Start at vertex A



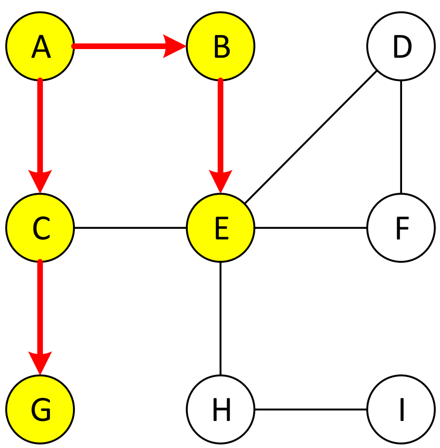
Explore vertices that are one-edge away from A.



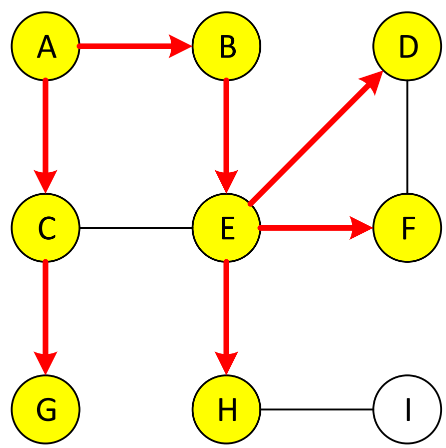
Graph Algorithms
BFS

- Illustration (continued)

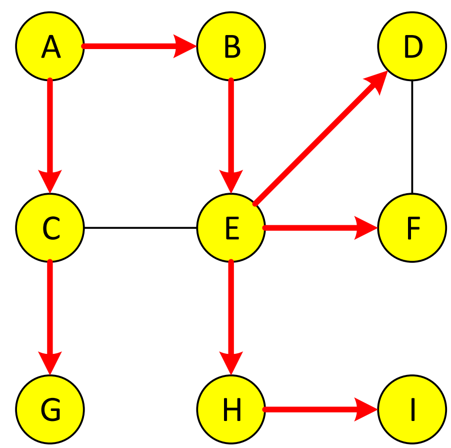
Explore vertices that are two-edge away from A.



Explore vertices that are three-edge away from A.



Explore vertices that are four-edge away from A. Finisahed



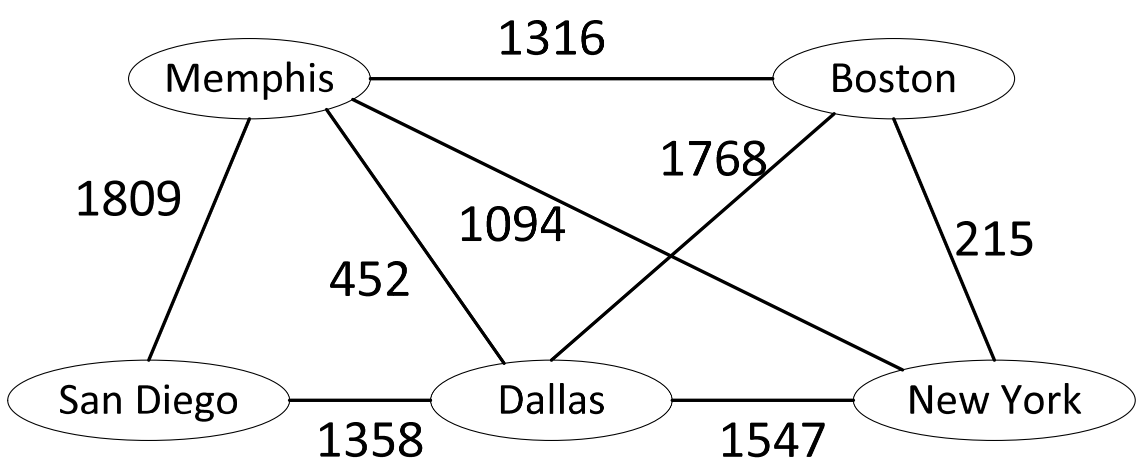
Graph Algorithms
BFS

- BFS properties:
  - The traversal visits all vertices reachable from *s*.
  - For each vertex *v* at level *i*, the path in the *BFS* tree from *s* to *v* has *i* edges, and any other path from s to *v* in *G* has at least *i* edges.
  - If (*u*, *v*) is an edge that is not in the *BFS* tree, the level number of *v* is at most 1 greater than the level number of *u*.

- Running time: $O(n + m)$

Graph Algorithms
Weighted Graph

- Each edge *e* is associated with a numeric label called *weight*, denoted *w*(*e*).
- Example



Graph Algorithms
Shortest Paths

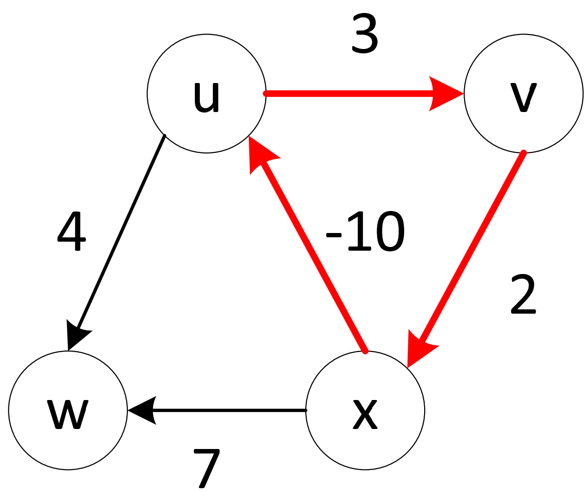
- Let G be a weighted graph.
  - The *length* of a path P is the sum of the weights of all edges on P. Let $P = <(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)>$. Then, the length of P, denoted $w(P)$, is defined as:

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

  - The *distance* from a vertex u to a vertex v in G, $d(u, v)$, is the length of a minimum-length path from u to v, if such path exists. The minimum-length path is referred to as *shortest path*.
  - $d(u, v) = \infty$, if there is no path from u to v in G.

Graph Algorithms
Shortest Paths

- Weights can be negative numbers. Then, a graph may have a *negative-weight cycle*:



- If a graph has a negative-weight cycle, a shortest path is not well defined.

Graph Algorithms
Dijkstra's Algorithm

- A well-known single-source shortest path algorithm on a directed or undirected graph G without negative weights.
- Finds shortest paths from a source vertex to every other vertex in G.
- A greedy algorithm.

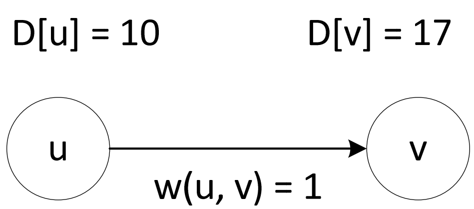
- Edge relaxation
  - $D[v]$ is the length of the best path from s to v we have found so far.
  - Initially $D[s] = 0$ and $D[v] = \infty$ for all other vertexes.
  - During the execution of the algorithm, $D[v]$ is updated iteratively and becomes a shortest-path length from s to v.

Graph Algorithms
Dijkstra's Algorithm

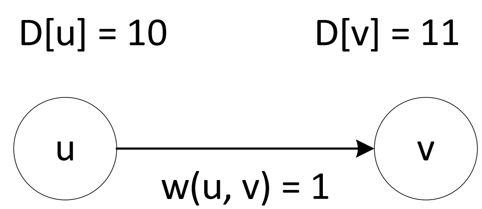
- Edge relaxation (continued)

if $D[u] + w(u, v) < D[v]$ then
  $D[v] = D[u] + w(u, v)$

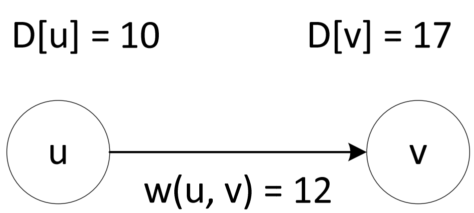
before relaxation



after relaxation



not relaxed



Graph Algorithms
Dijkstra's Algorithm

- Pseudocode

Algorithm ShortestPath(*G*, *s*):

Input: A directed or undirected graph G with nonnegative weights, and a distinguished vertex *s* of G

Output: The length of a shortest path from *s* to *v* for every vertex *v* of *G*

Ininialize D[s] = 0 and D[v] = ∞ for each vertex $v \neq s$

Let a priority queue *Q* contains all vertices of *G* using *D* labels as keys

while *Q* is not empty do

  u = Q.removeMin( ) // vertex with the smallest D[u] is pulled into "cloud"

  for each edge (*u*, *v*) such that *v* is in *Q* do

    // perform relaxation

    if $D[u] + w(u, v) < D[v]$ then

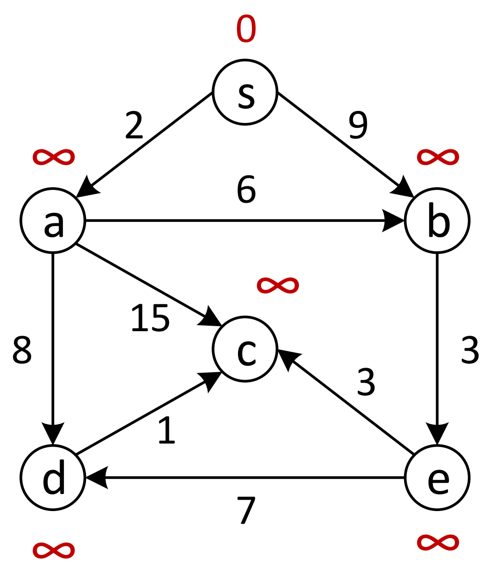
      $D[v] = D[u] + w(u, v)$

      Change the key of vertex *v* in *Q* to $D[v]$

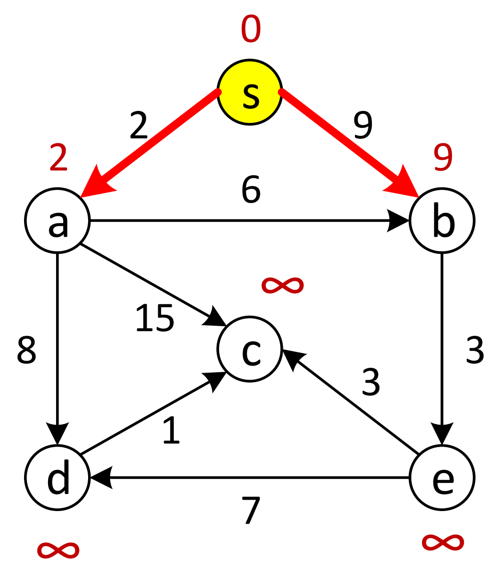
return the label $D[v]$ of each vertex *v*

Graph Algorithms
Dijkstra's Algorithm

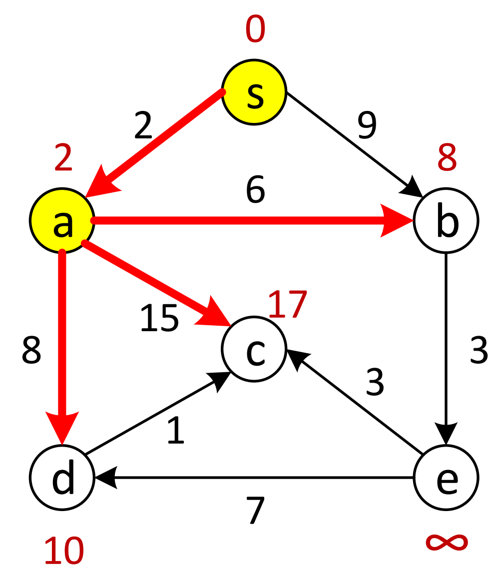
- Illustration



(a) Initially, all vertices are in Q, C is empty, D[s] = 0, D[v] = ∞ for all other vertices.



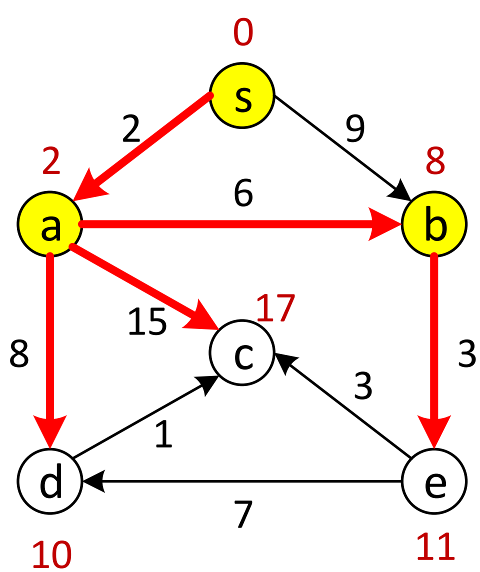
(b) s comes into C, edges (s, a) and (s, b) are relaxed.



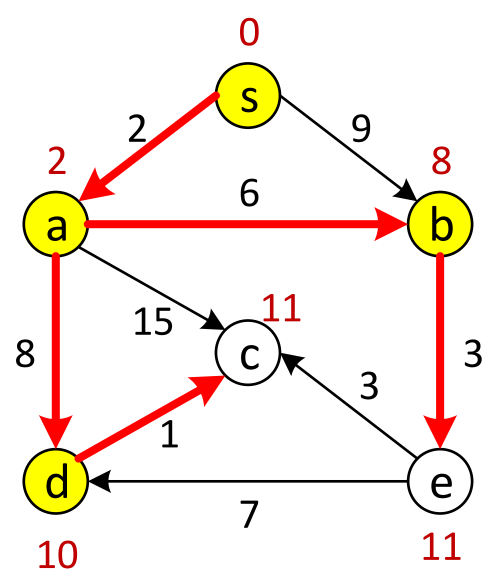
(c) a comes into C, edges (a, b), (a, c), and (a, d) are relaxed.

Graph Algorithms
Dijkstra’s Algorithm

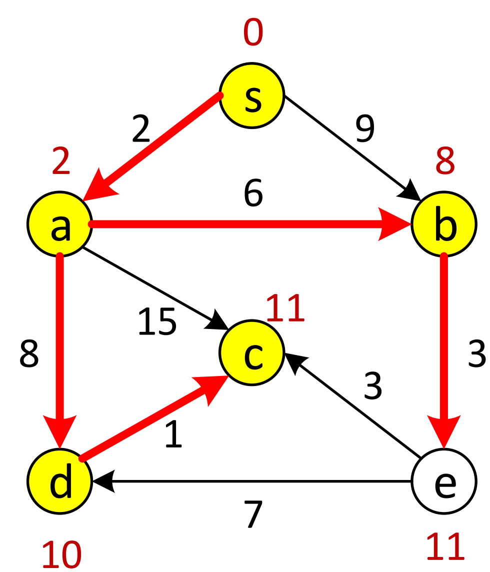
- Illustration (continued)



(d) b comes into C, edge (b, e) is relaxed.



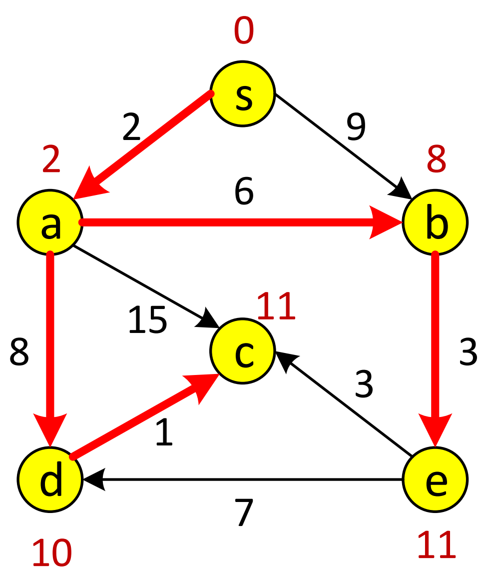
(e) d comes into C, edge (d, c) is relaxed.



(f) c comes into C. No edge relaxation needed.

Graph Algorithms
Dijkstra's Algorithm

- Illustration (continued)



(g) e comes into C. No edge relaxation needed. Finished.

Running time: $O((n + m) \log n)$

Graph Algorithms
Minimum Spanning Trees

- Given a tree T in an undirected, weighted graph G, the weight of T, $w(T)$, is defined as follows:

$$w(T) = \sum_{(u,v)\ in\ T} w(u,v)$$

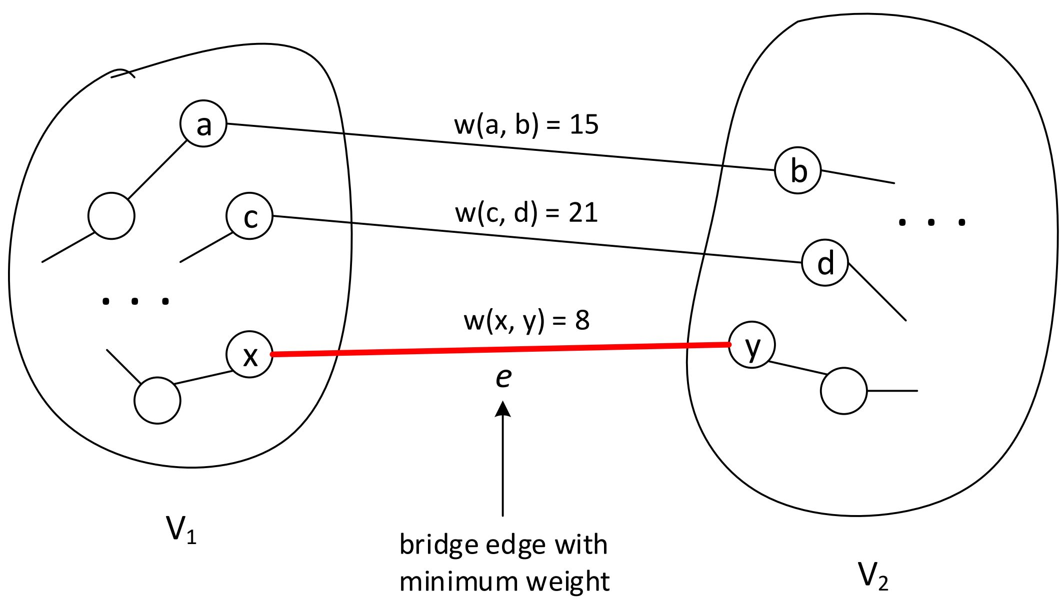
- A *minimum spanning tree* of an undirected, weighted graph G is a spanning tree with the minimum weight.
- Minimum spanning tree problem: Find such a tree in G.
- Will discuss two algorithms, Prim-Jarnik algorithm and Kruskal's algorithm, both of which are greedy algorithms.

- We assume that a graph G is undirected, weighted, connected, and simple.

Graph Algorithms
Minimum Spanning Trees

- Bridge edge and minimum-weight (bridge) edge

- Suppose *G* is partitioned into mutually exclusive V_1 and V_2.
- Bridge edge: one end in V_1 and the other in V_2.
- Minimum-weight edge: a bridge with the smallest weight



Graph Algorithms
Prim-Jarnik Algorithm

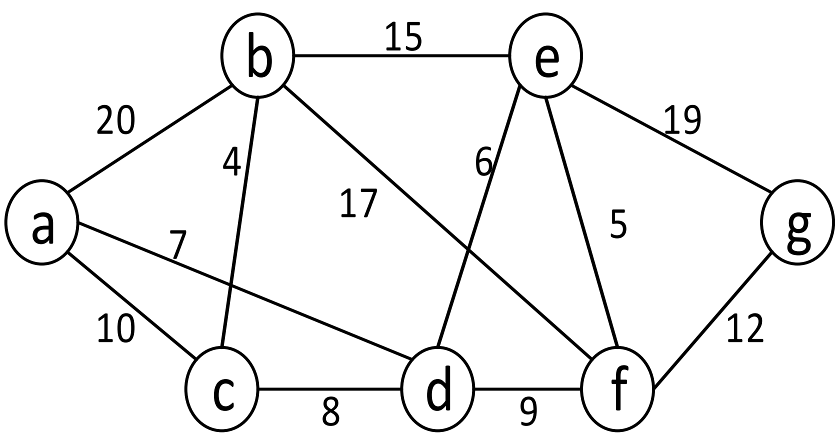
- Outline
  - Begins at some “root” vertex *s*.
  - Keeps a set of vertices *C*, called “cloud.”
  - Initially, *C* has only *s*.
  - In each iteration, we find a minimum-weight edge connecting a vertex *u* in the cloud of *C* and a vertex *v* that is outside the cloud.
  - Then, the vertex *v* is pulled into *C*
  - This process is repeated until a spanning tree is formed.

Graph Algorithms
Prim-Jarnik Algorithm

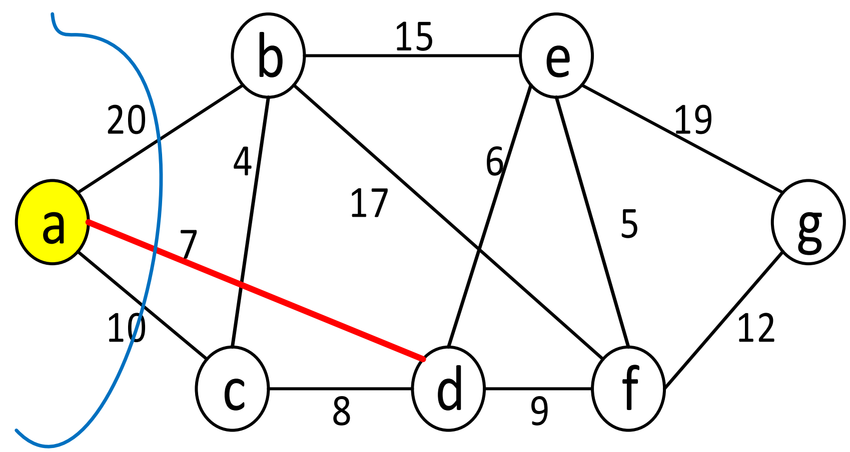
- Outline (continued)
  - Each vertex *v* has a label *D*[*v*], which stores the weight of the minimum observed edge connecting *v* to the cloud *C*.
  - Vertices that are not in C are stored in a priority queue, where *D*[*v*] is used as a key in the queue.
  - If we choose a vertex in the priority queue with the minimum *D*[*v*], then it is a minimum-weight edge.

Graph Algorithms
Prim-Jarnik Algorithm

- Illustration
  - Red: min-weight edge;
  - Yellow: in the cloud
  - Blue: cloud boundary



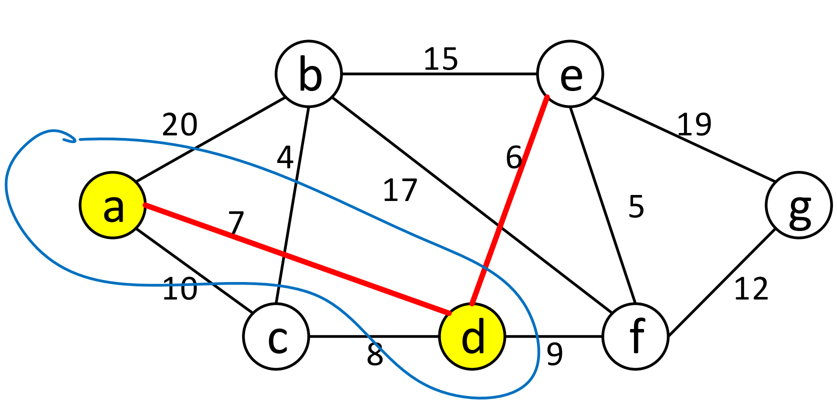
(a) Initial tree. Begin at a



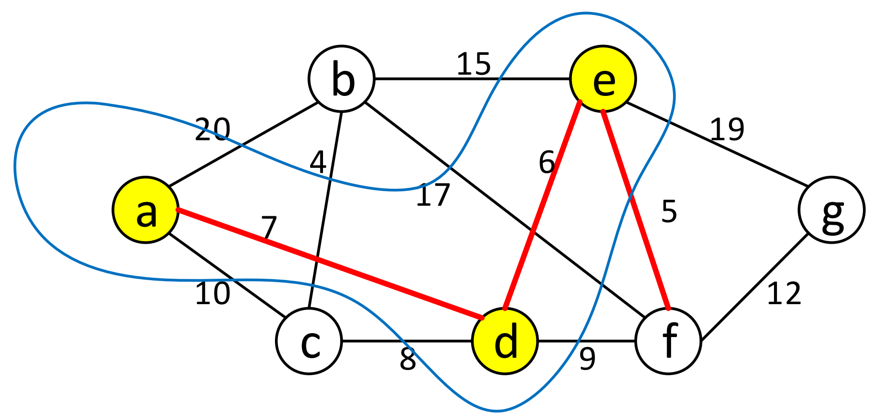
(b) (a, d) is minimum-weight edge

Graph Algorithms
Prim-Jarnik Algorithm

- Illustration (continued)



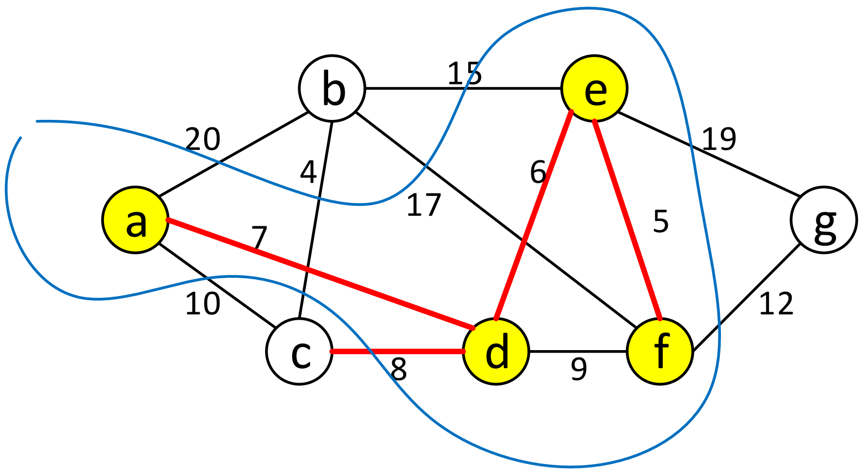
(c) (d, e) is minimum-weight edge



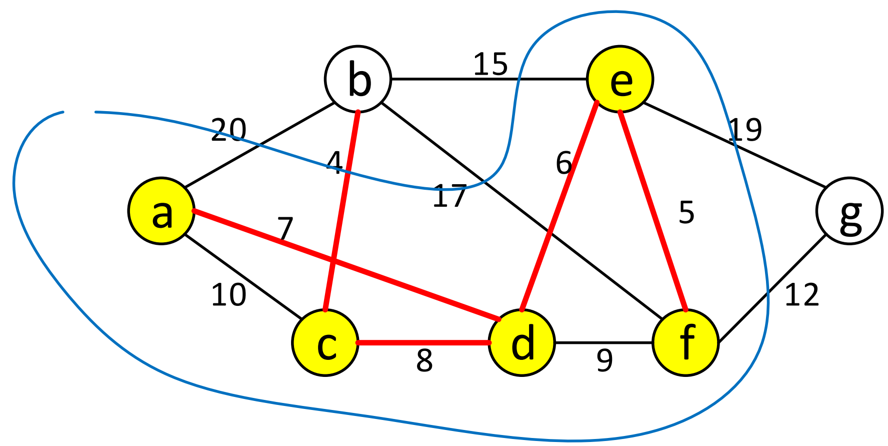
(d) (e, f) is minimum-weight edge

Graph Algorithms
Prim-Jarnik Algorithm

- Illustration (continued)



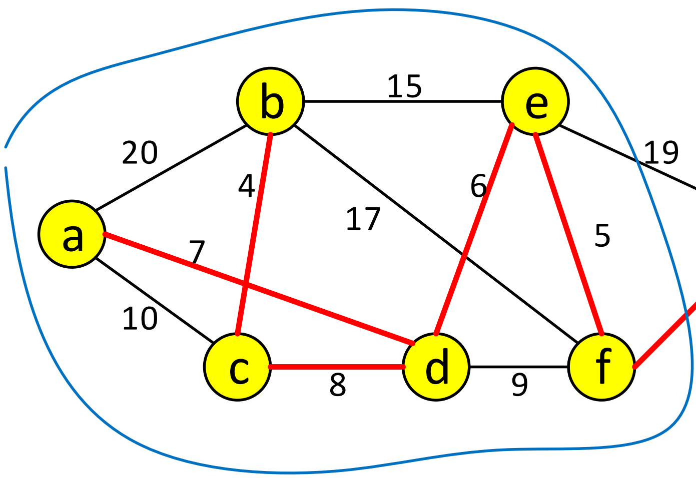
(e) (c, d) is minimum-weight edge



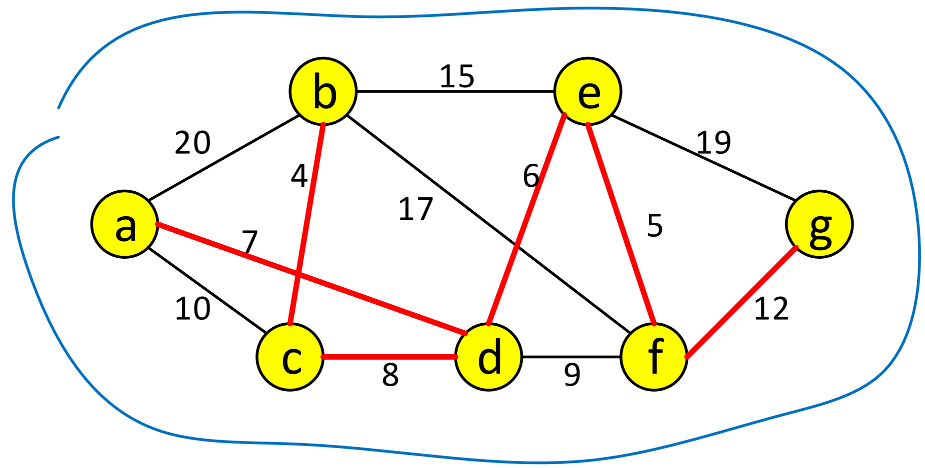
(f) (b, c) is minimum-weight edge

Graph Algorithms
Prim-Jarnik Algorithm

- Illustration (continued)



(g) (f, g) is minimum-weight edge



(h) Finished. Thick red edges form MST

- Running time: $O((n + m) \log n)$

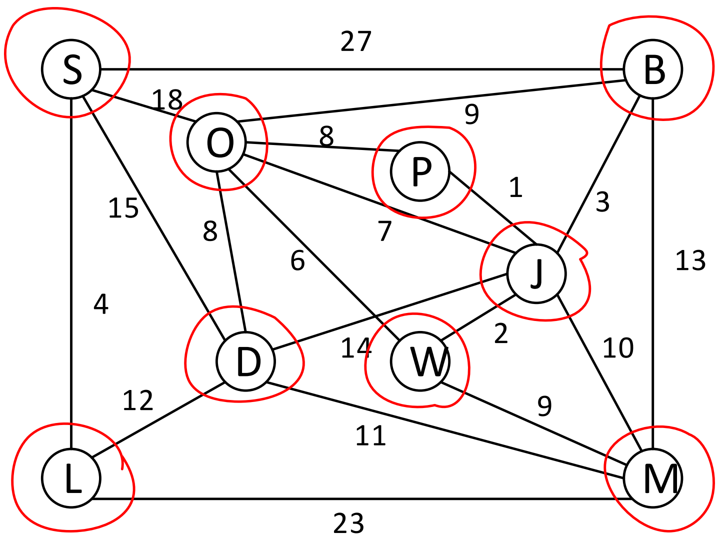
Graph Algorithms
Kruskal's Algorithm

- In the Prim-Jarnik's algorithm, there is always a single tree.
- In the Kruskal's algorithm, there are multiple trees, all of which are eventually merged into an MST.
- Outline: Initially, a spanning tree *T* is empty and each vertex is a "cluster" on its own.

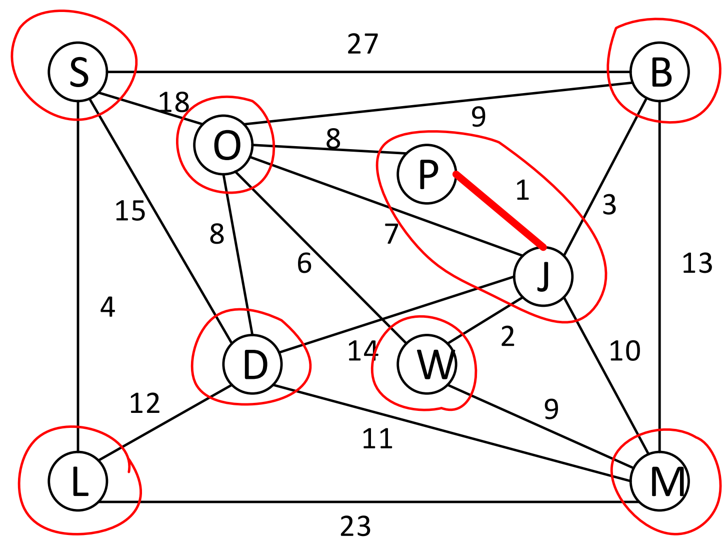
  - Step 1: Find an edge *e* with the smallest weight.
  - Step 2: If two endpoints of *e* belong to different clusters, merge those two clusters.
  - Step 3: Include *e* in *T*.
  - Step 4: Stop if all vertices are included by *T*. Otherwise, return to Step 1 and repeat.

Graph Algorithms
Kruskal's Algorithm

- Pseudocode.
- Illustration



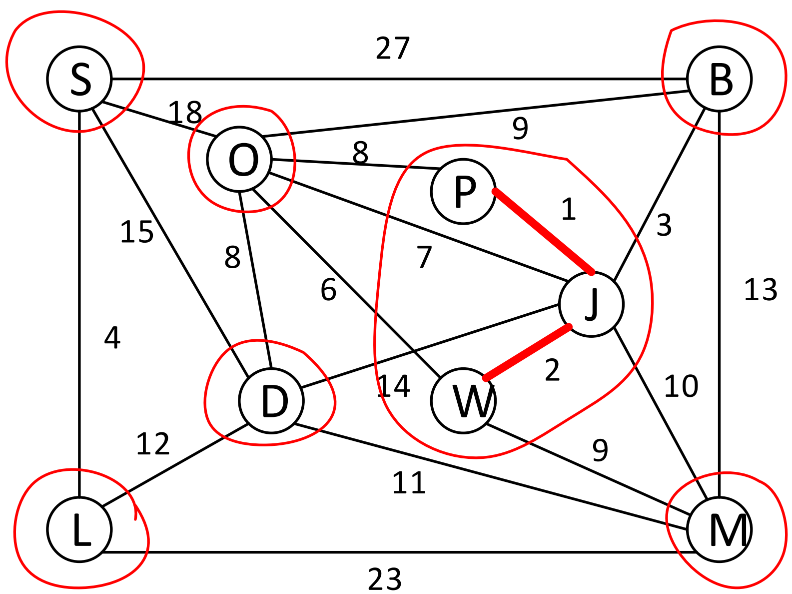
(a) Initial tree. Each vertex is its own cluster. w(J,P) is the smallest.



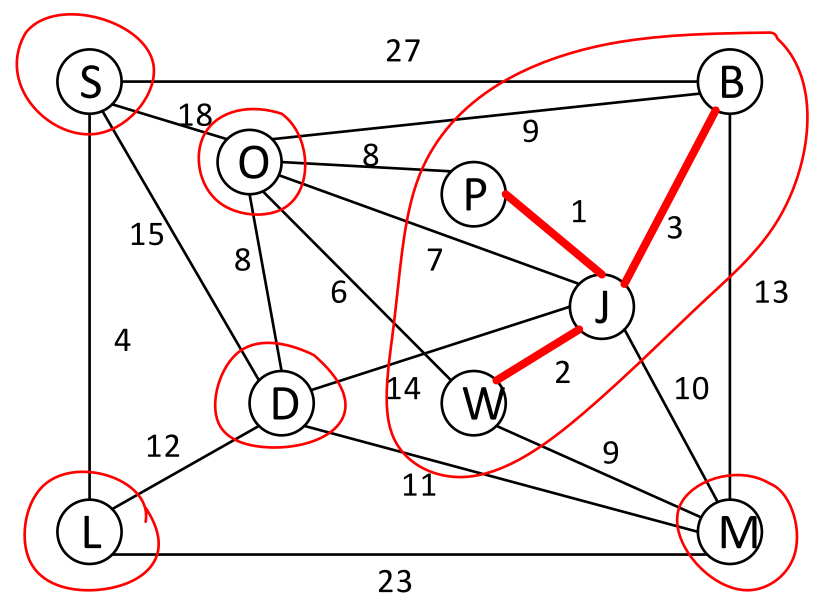
(b) w(J,W) is the next smallest.

Graph Algorithms
Kruskal's Algorithm

- Illustration (continued)



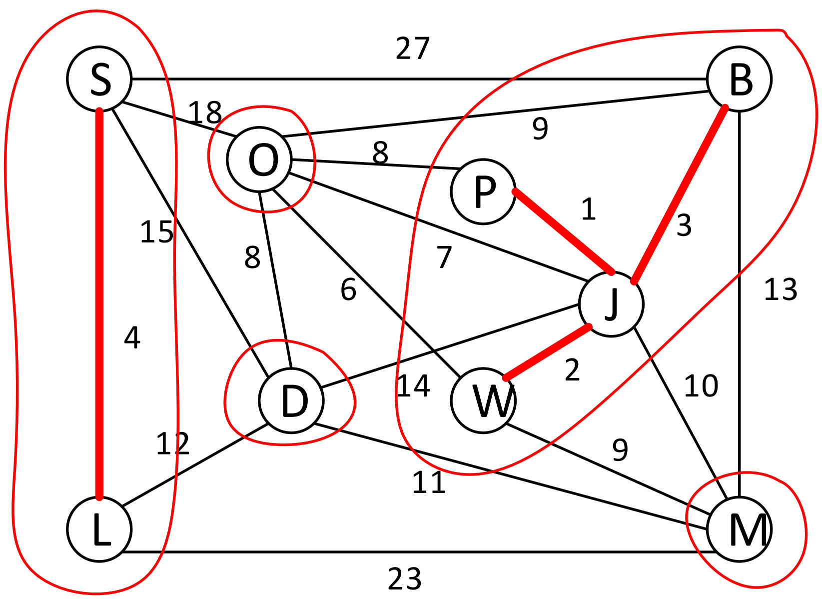
(c) w(B,J) is the next smallest.



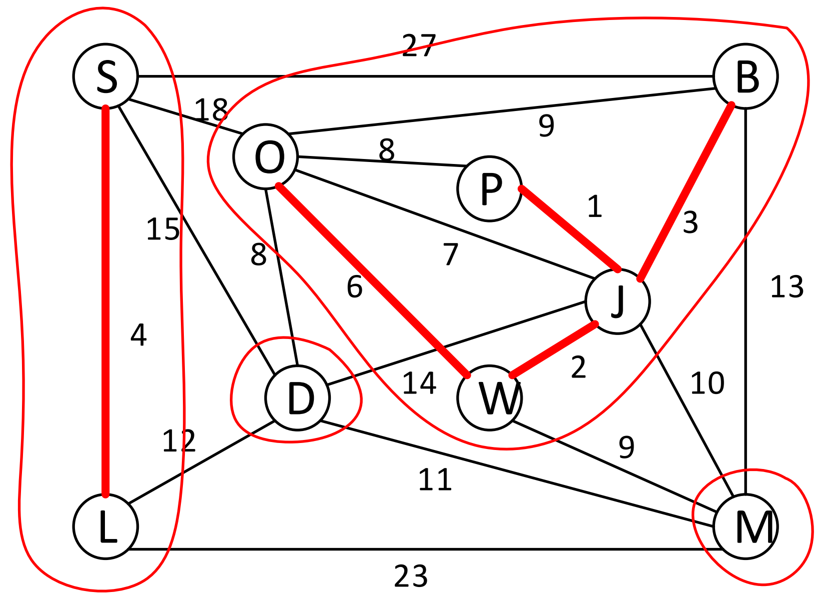
(d) w(L,S) is the next smallest.

Graph Algorithms
Kruskal's Algorithm

- Illustration (continued)



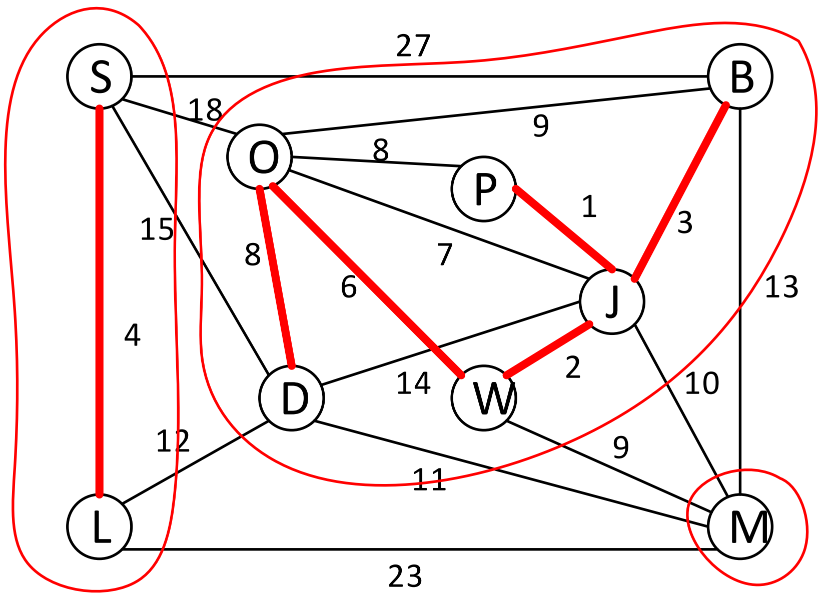
(e) w(O,W) is the next smallest.



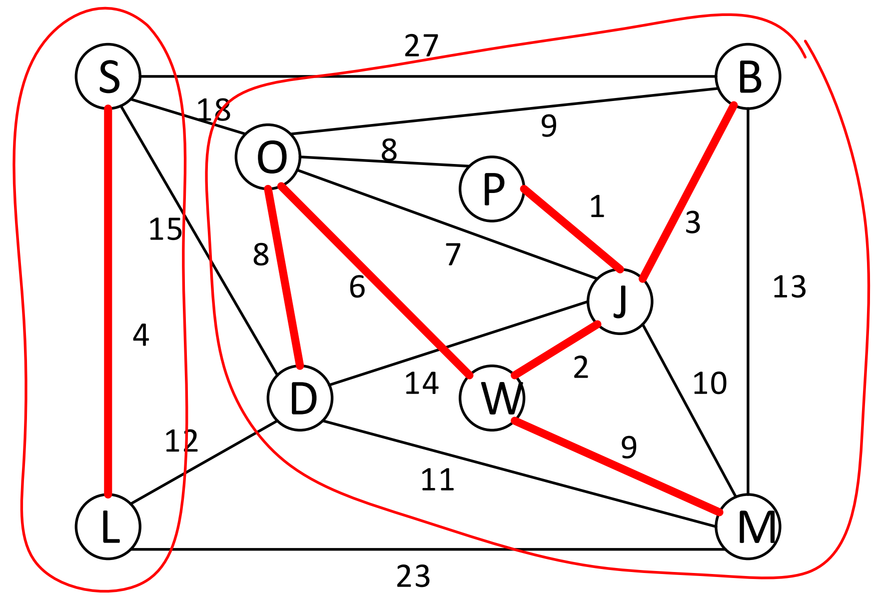
(f) w(J,O) is the next smallest. But, they are in the same cluster. w(O,P) the same. w(D,O) is the next smallest.

Graph Algorithms
Kruskal's Algorithm

- Illustration (continued)



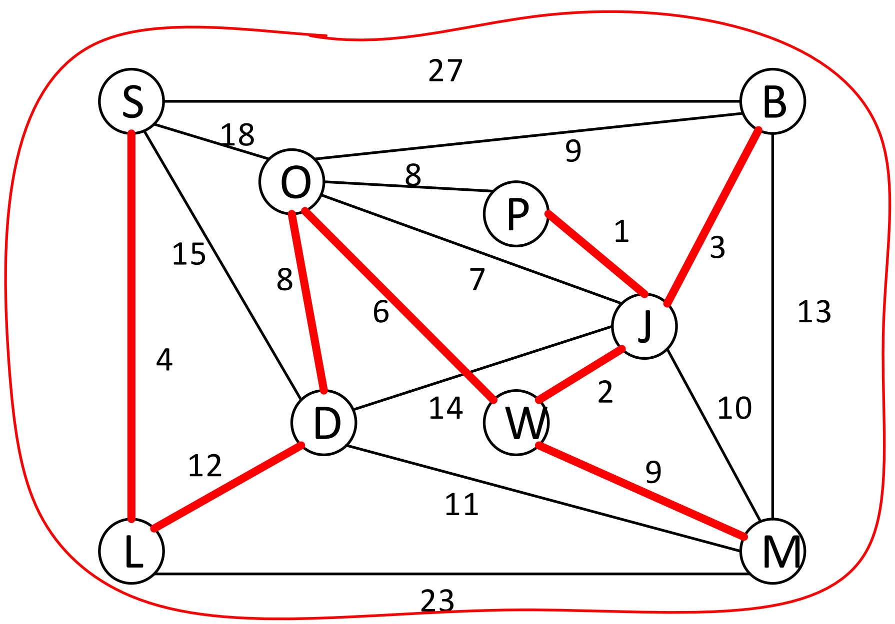
(g) w(B,O) is the next smallest. But, they are in the same cluster. w(M,W) is the next smallest.



(h) w(J,M) is the next smallest. But, they are in the same cluster. w(D,M) the same. w(D,L) is the next smallest.

Graph Algorithms
Kruskal's Algorithm

- Illustration (continued)



(i) Finished. Thick red edges form a minimum spanning tree.

Running time: $O(m \log n)$

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.