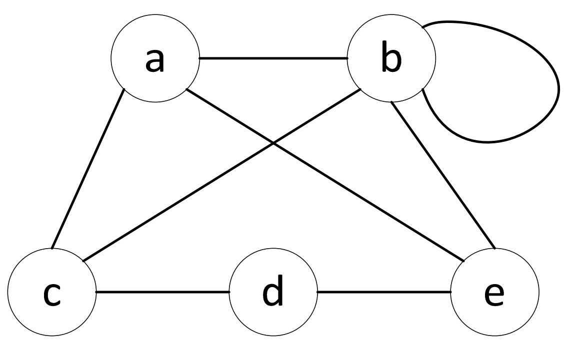
Data Structures and Algorithms

Week 6

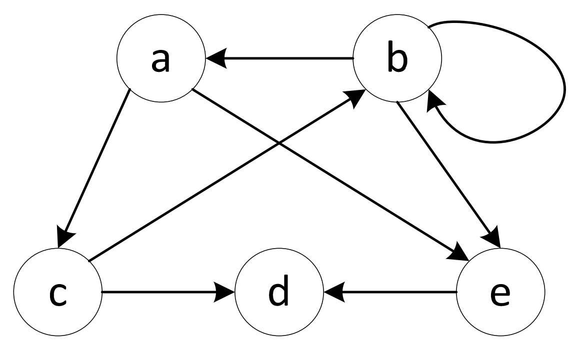
Graph Algorithms
Basics

- 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

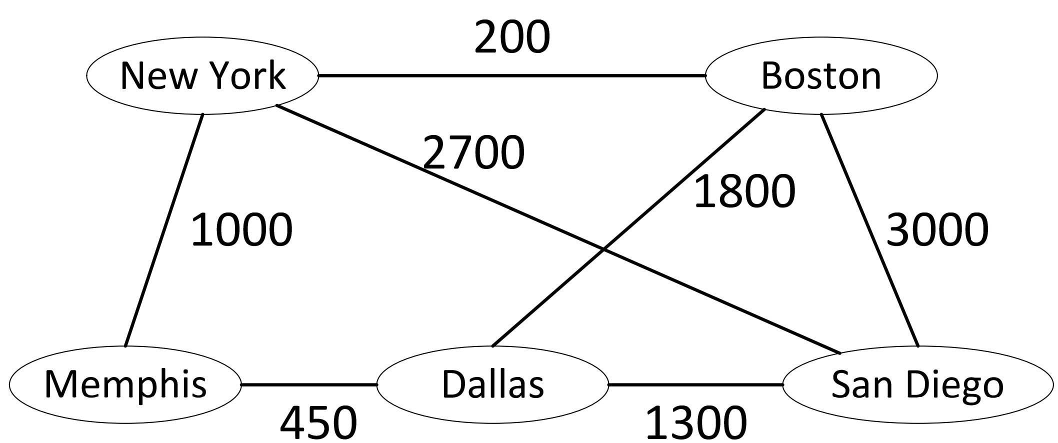


(b) Directed graph



Graph Algorithms
Basics

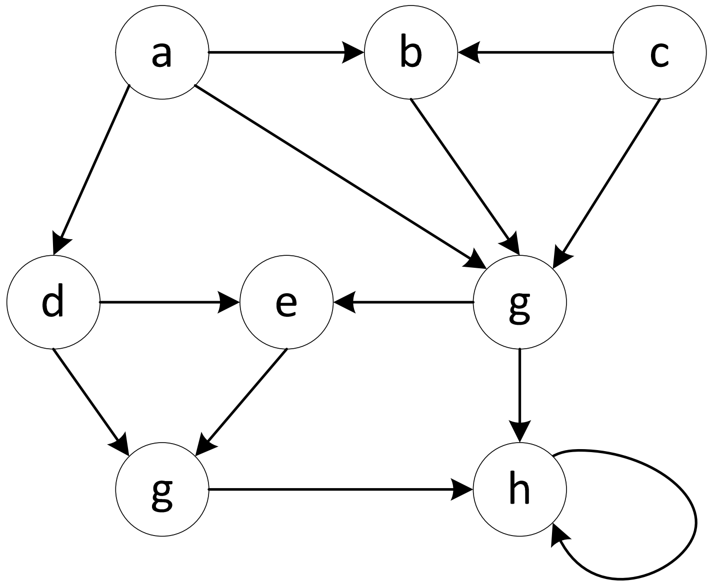
- Two vertices *u* and *v* are said to be *adjacent* if there is an edge (*u*, *v*).
- An edge is said to be *incident* to a vertex if the vertex is one of the edge's endpoints.
- *Weighted* graph: An information (usually called weight) is associated with edges



Graph Algorithms
Basics

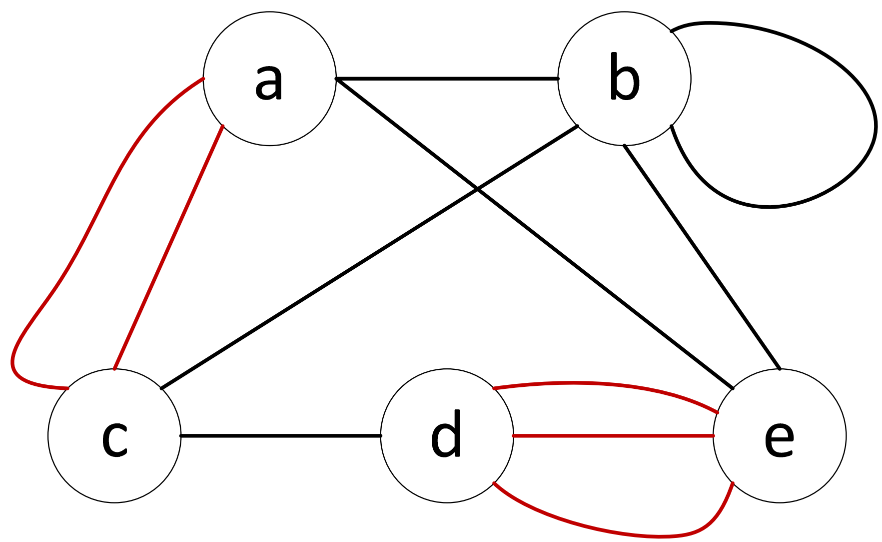
- Outgoing edge vs. incoming edge
- Degree, in-degree, and out-degree of a node

- 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

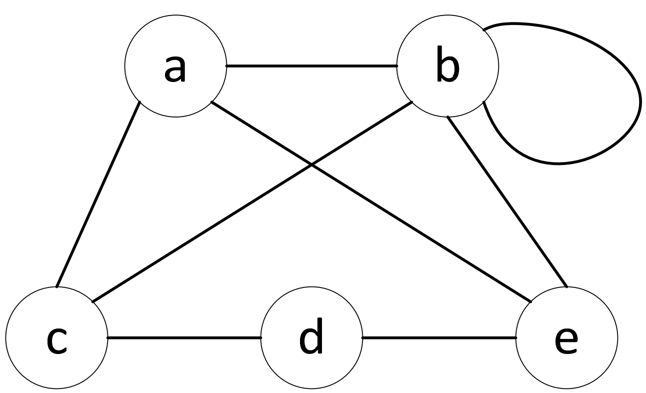
- Parallel edges and self-loops



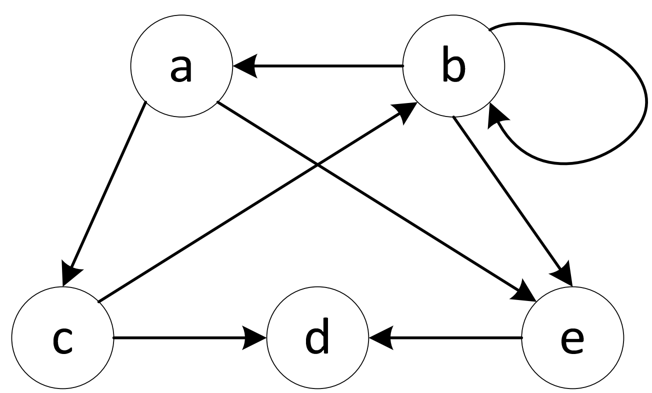
Graph Algorithms
Basics

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

(a) Undirected graph

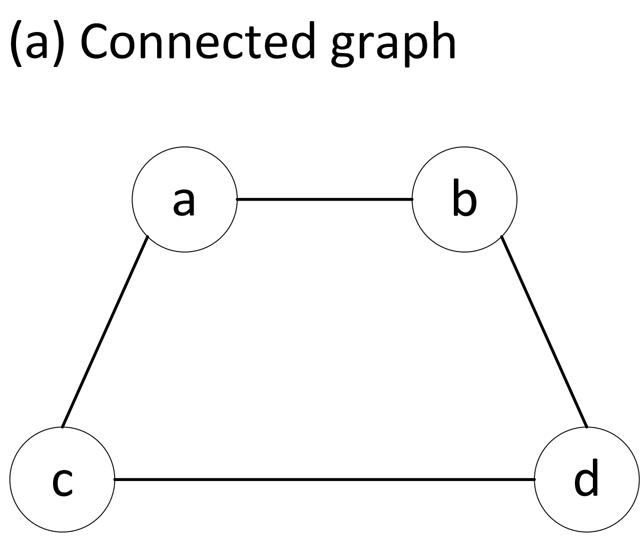


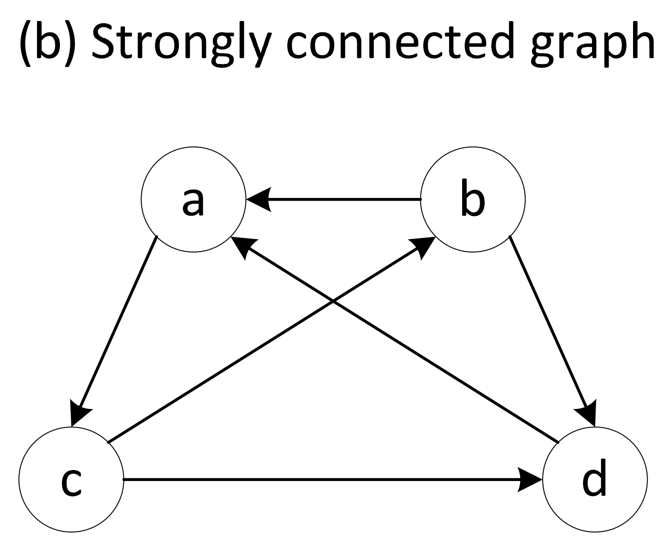
(b) Directed graph



Graph Algorithms
Basics

- *Connected graph* and *strongly connected graph*

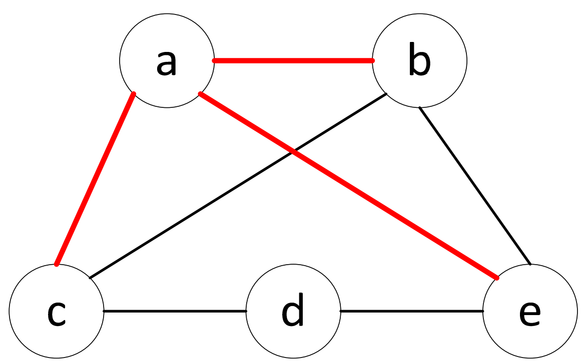




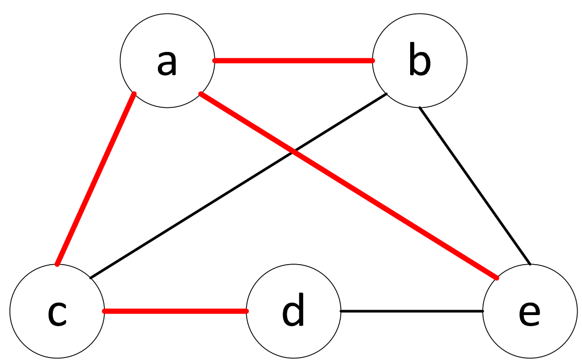
Graph Algorithms
Basics

- *Subgraph* and *spanning subgraph*

(a) A subgraph



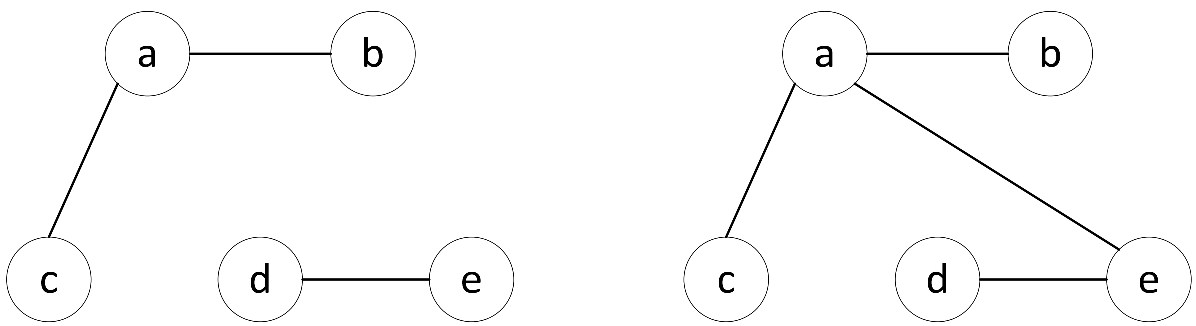
(b) A spanning subgraph



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

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

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

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

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

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

- Opr erations
  - 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*)

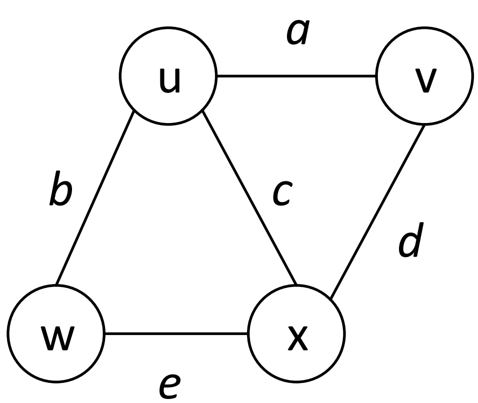
Graph Algorithms
Data Structures for Graphs

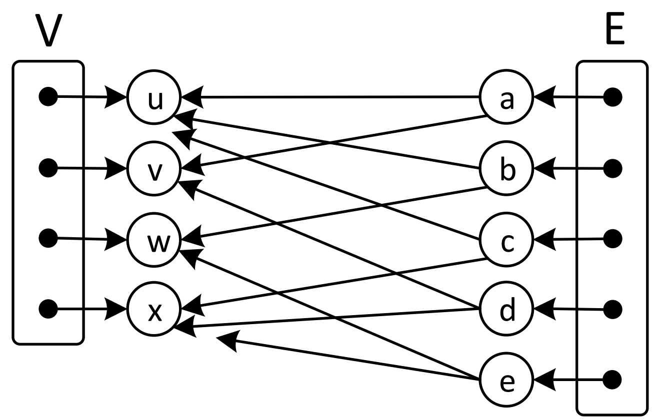
- Edge list, adjacency list, adjacency map, adjacency matrix

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

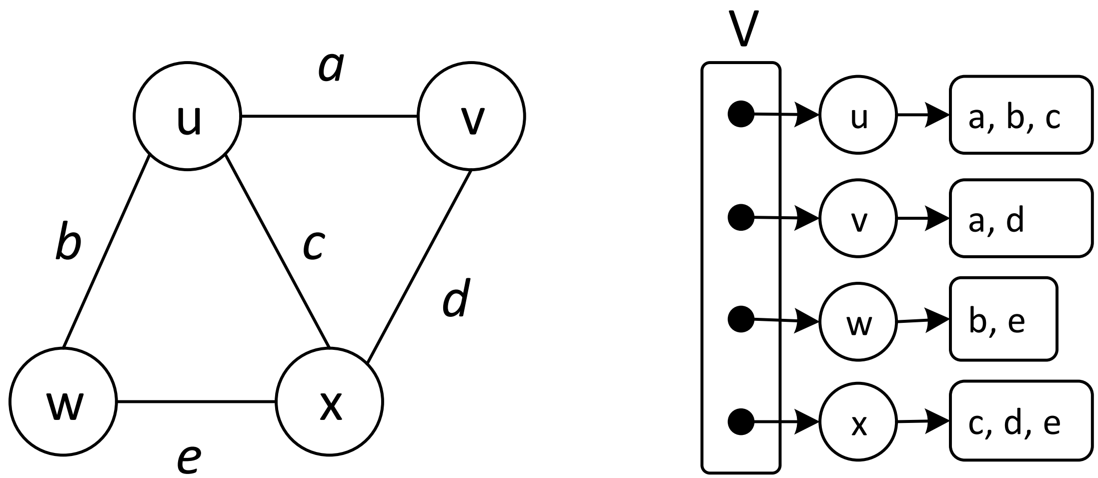




- *V* is a list of vertices and *E* is a list of edges. Both can be implemented using doubly linked lists.

Graph Algorithms
Data Structures for Graphs

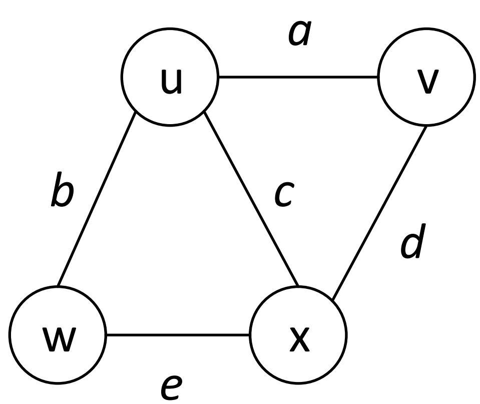
- Adjacency list

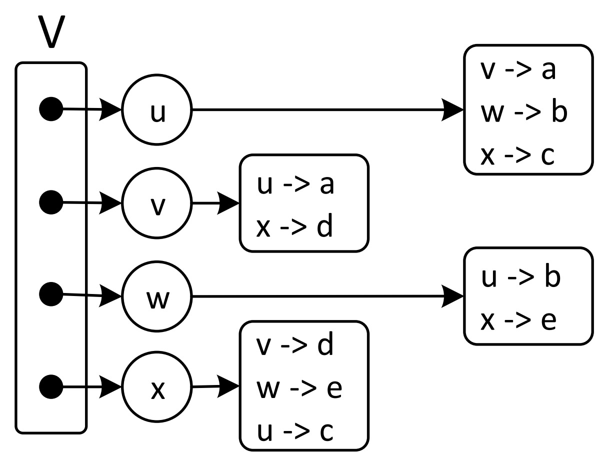


- *V* is a list of vertices.
- Each vertex *v* has a reference to a separate collection of edges that are incident to *v*.
- The collection is called *incidence collection*.

Graph Algorithms
Data Structures for Graphs

- Adjacency map

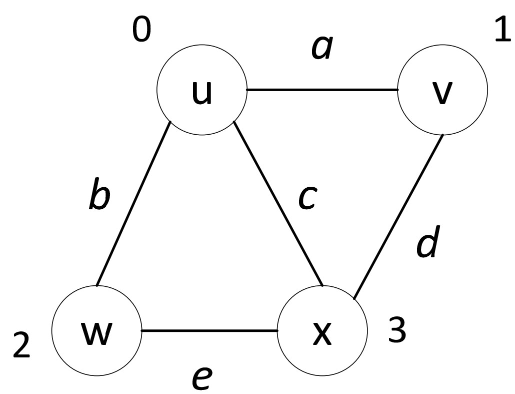




- Incidence collection of *V* is implemented as a map.
- Suppose edge *a* = (*u*, *v*) is in the incidence collection of *u*. Then, <*v*, *a*> pair is stored in the map, where *v* is a key and *a* is the corresponding value.

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

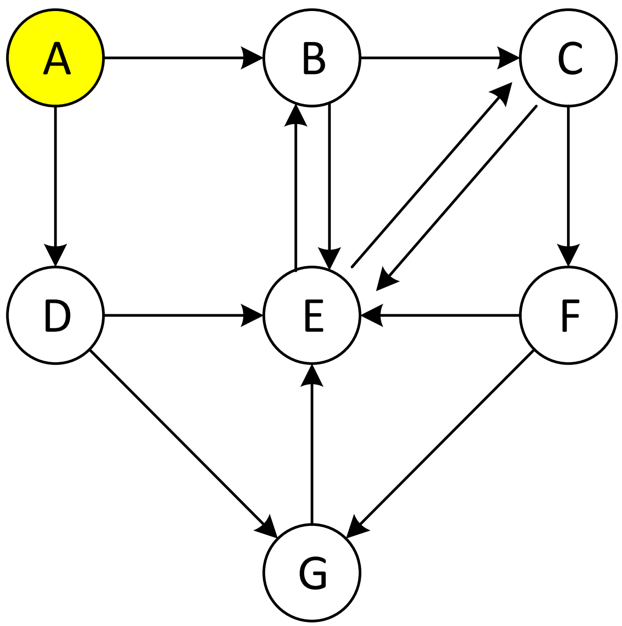
- Pseudocode

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)

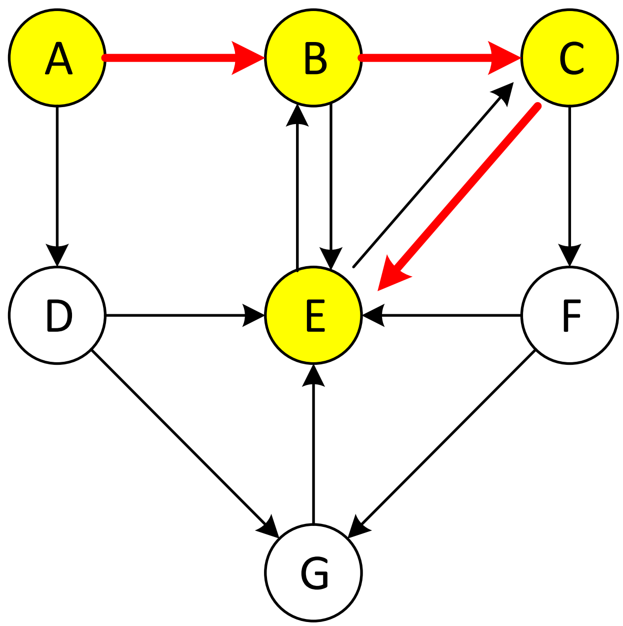
Graph Algorithms
DFS

- Illustration (on a directed graph)

A directed graph
Start at vertex A



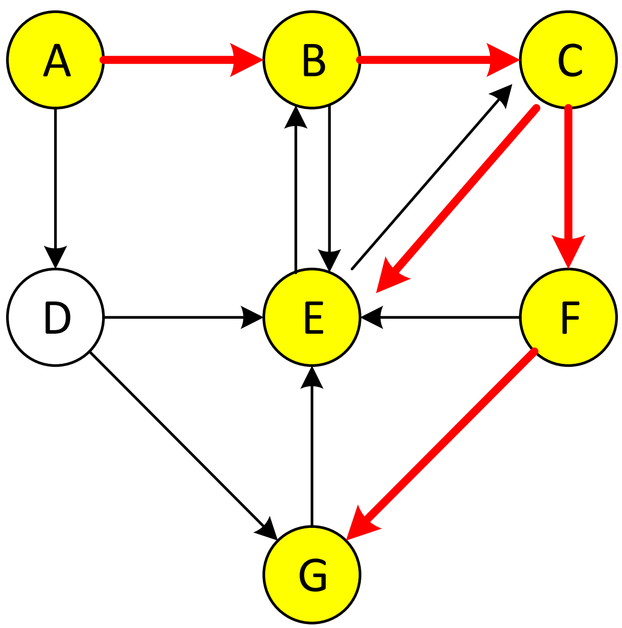
A -> B -> C -> E
Backtrack to C



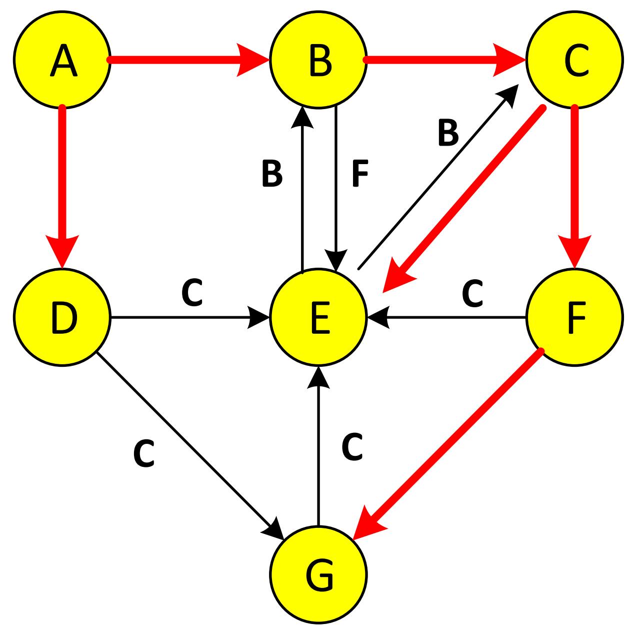
Graph Algorithms
DFS

- Illustration (on a directed graph)

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



A -> D
Finished



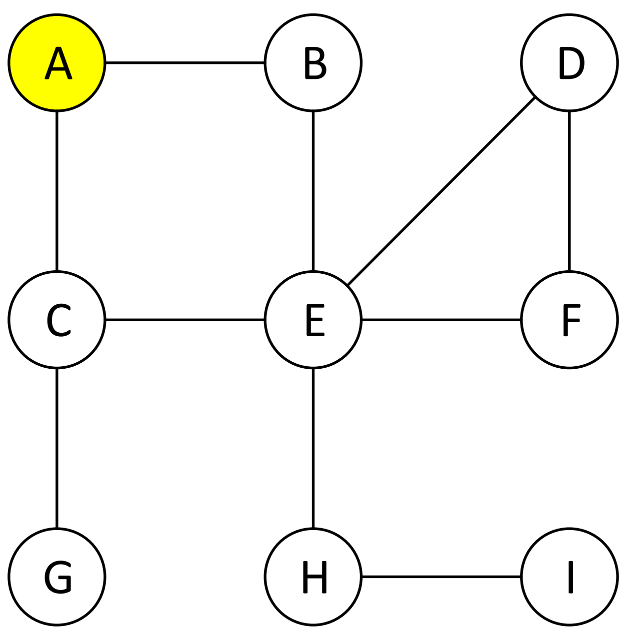
Graph Algorithms
DFS

- 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*.

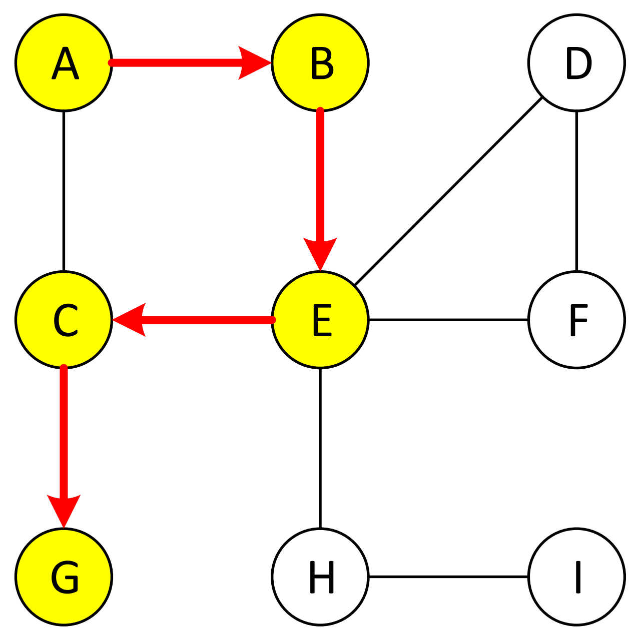
Graph Algorithms
DFS

- Illustration (on an undirected graph)

An undirected graph
Start at vertex A



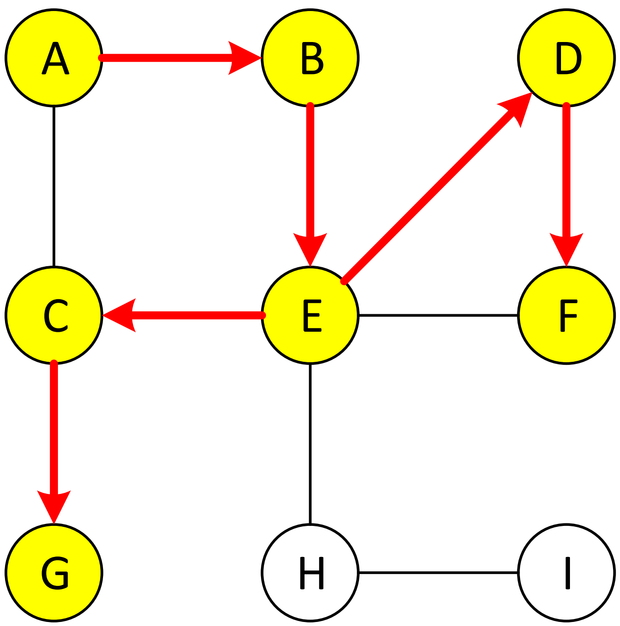
A -> B -> E -> C -> G
Backtrack to C -> E



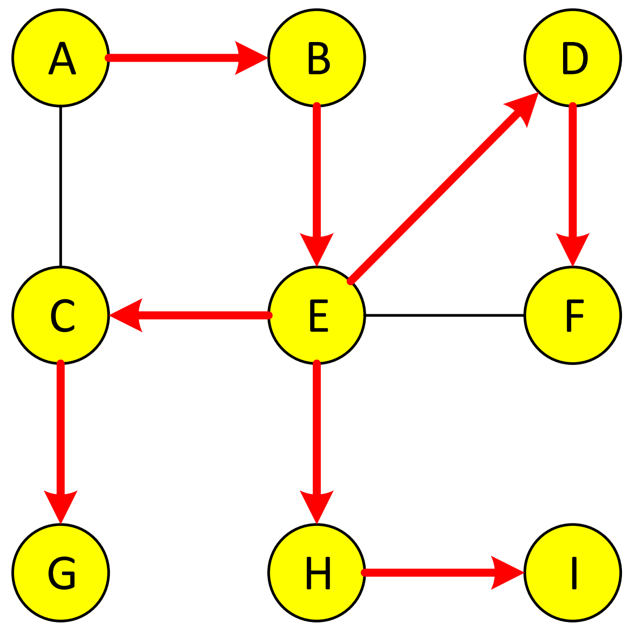
Graph Algorithms
DFS

- Illustration (on an undirected graph)

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



E -> H -> I
Finished



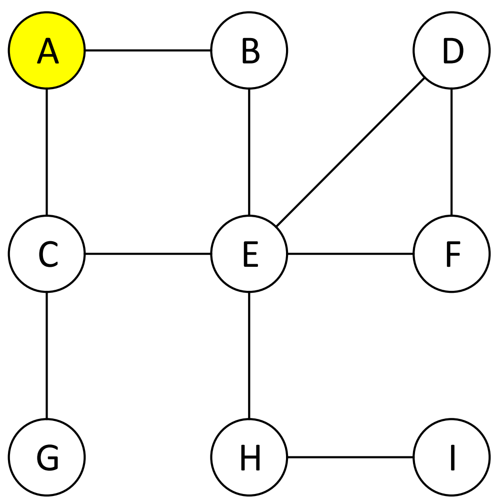
Graph Algorithms
DFS

- 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

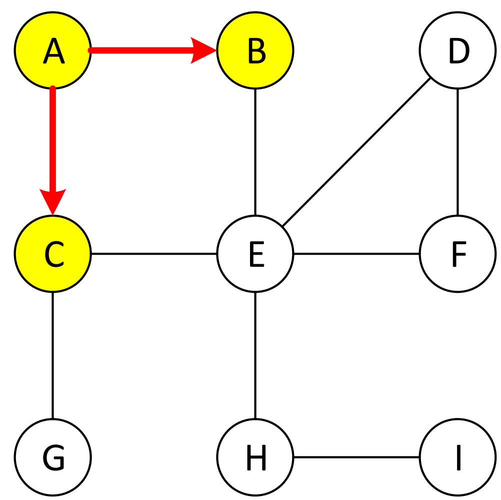
Graph Algorithms
BFS

- 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



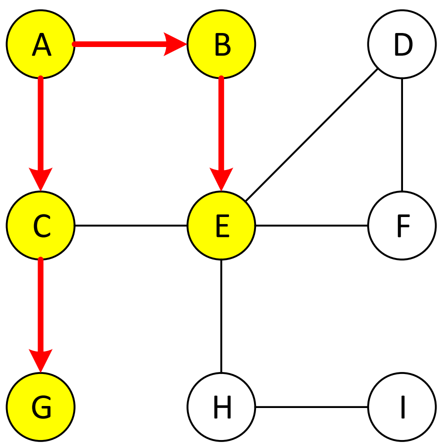
Explore vertices that are one-edge away from A.



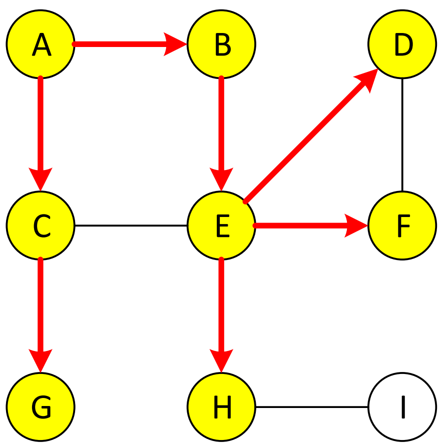
Graph Algorithms
BFS

- Illustration (continued)

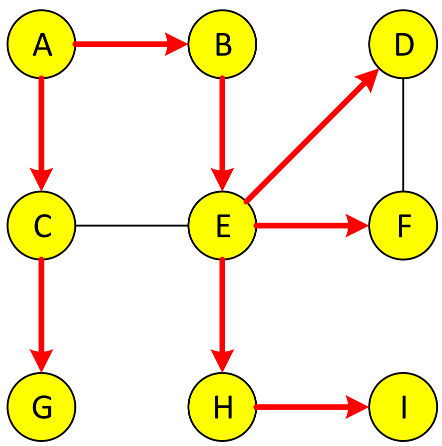
Explore vertices that are two-edge away from A.



Explore vertices that are three-edge away from A.



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



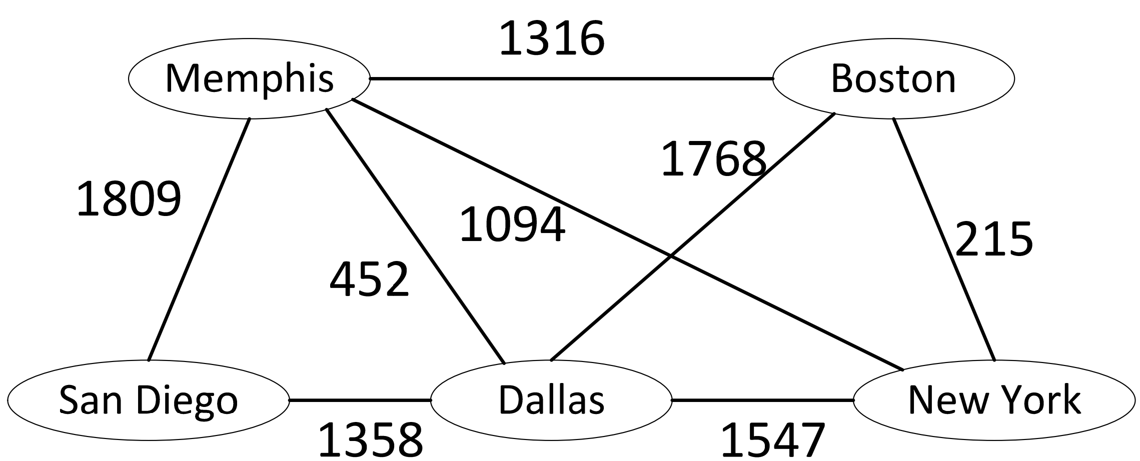
Graph Algorithms
BFS

- 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

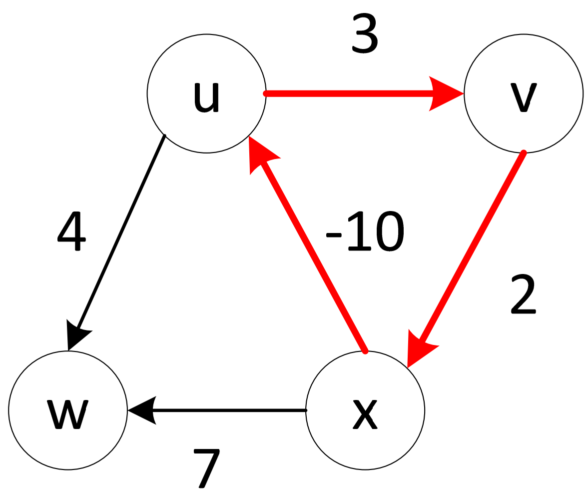
- Let G be a weighted graph.
  - The *length* of a path P is the sum of the weights of all edges on P. Let $P = <(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)>$. Then, the length of P, denoted $w(P)$, is defined as:

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

  - 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.

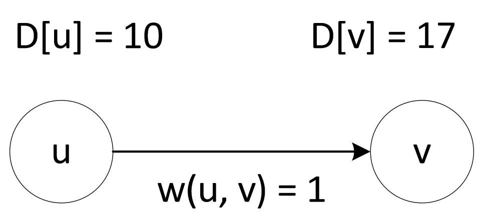
- Edge relaxation
  - *D*[*v*] is the length of the best path from *s* to *v* we have found so far.
  - Initially *D*[*s*] = 0 and *D*[*v*] = ∞ for all other vertexes.
  - During the execution of the algorithm, *D*[*v*] is updated iteratively and becomes a shortest-path length from *s* to *v*.

Graph Algorithms
Dijkstra's Algorithm

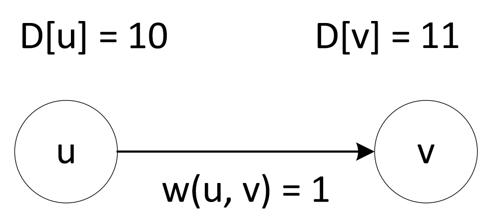
- Edge relaxation (continued)

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

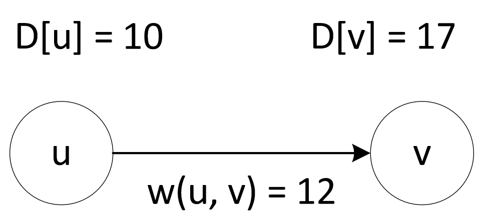
before relaxation



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*

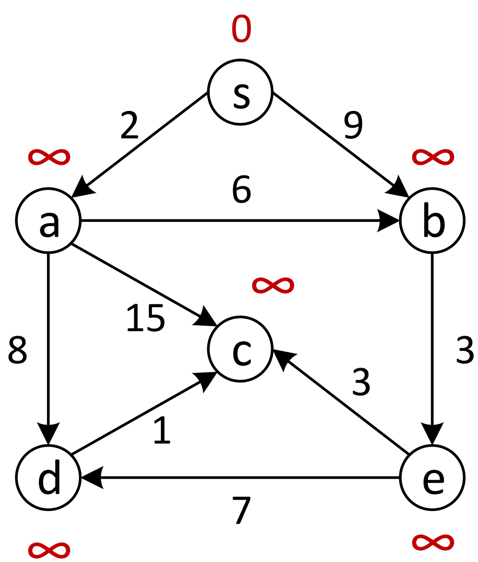
Ininialize D[s] = 0 and D[v] = ∞ for each vertex $v \neq s$

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

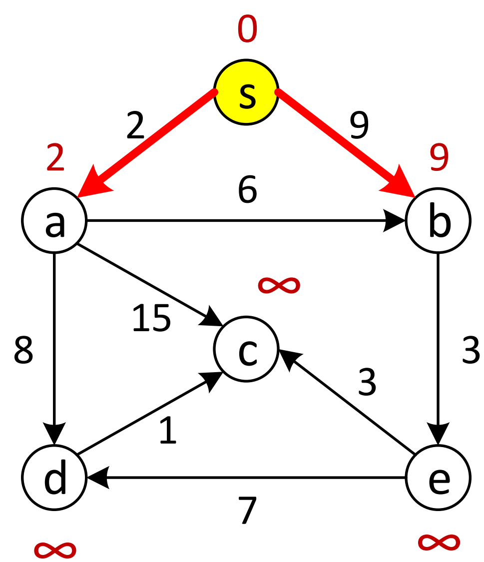
```
while Q is not empty do
  u = Q.removeMin( ) // vertex with the smallest D[u] is pulled into "cloud"
  for each edge (u, v) such that v is in Q do
    // perform relaxation
    if D[u] + w(u, v) < D[v] then
      D[v] = D[u] + w(u, v)
      Change the key of vertex v in Q to D[v]
return the label D[v] of each vertex v
```

Graph Algorithms
Dijkstra's Algorithm

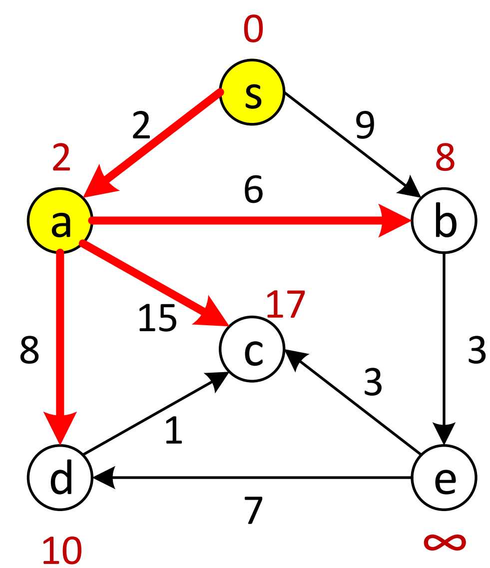
- Illustration



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



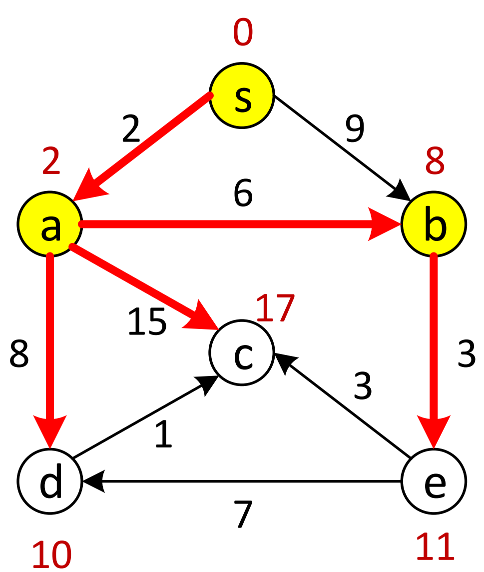
(b) s comes into C, edges (s, a) and (s, b) are relaxed.



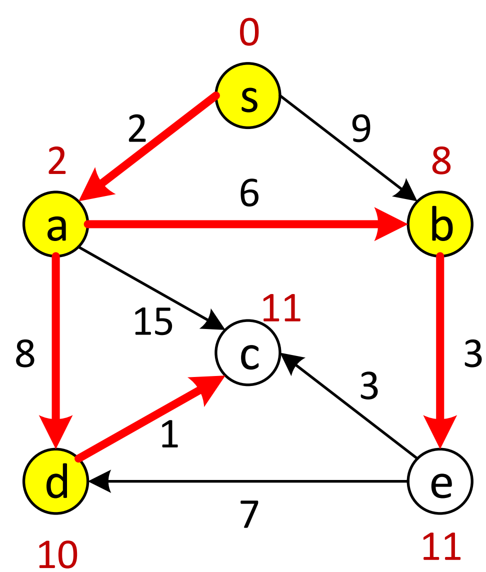
(c) a comes into C, edges (a, b), (a, c), and (a, d) are relaxed.

Graph Algorithms
Dijkstra's Algorithm

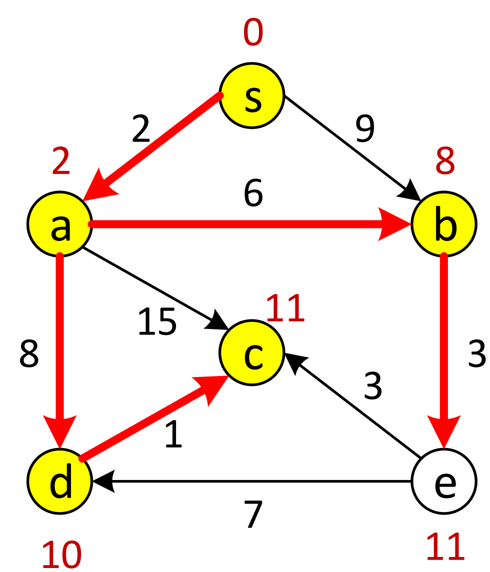
- Illustration (continued)



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



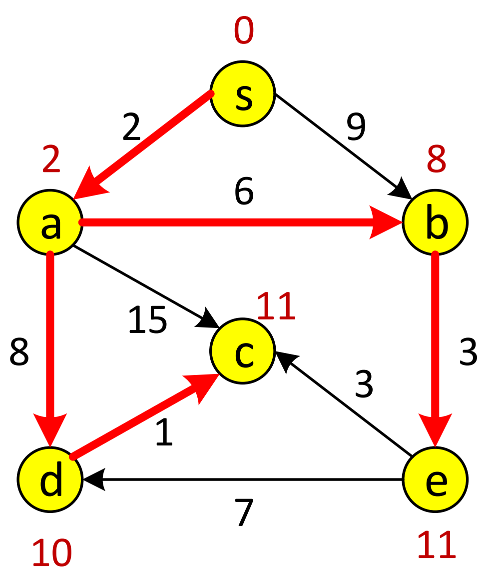
(e) d comes into C, edge (d, c) is relaxed.



(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:

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

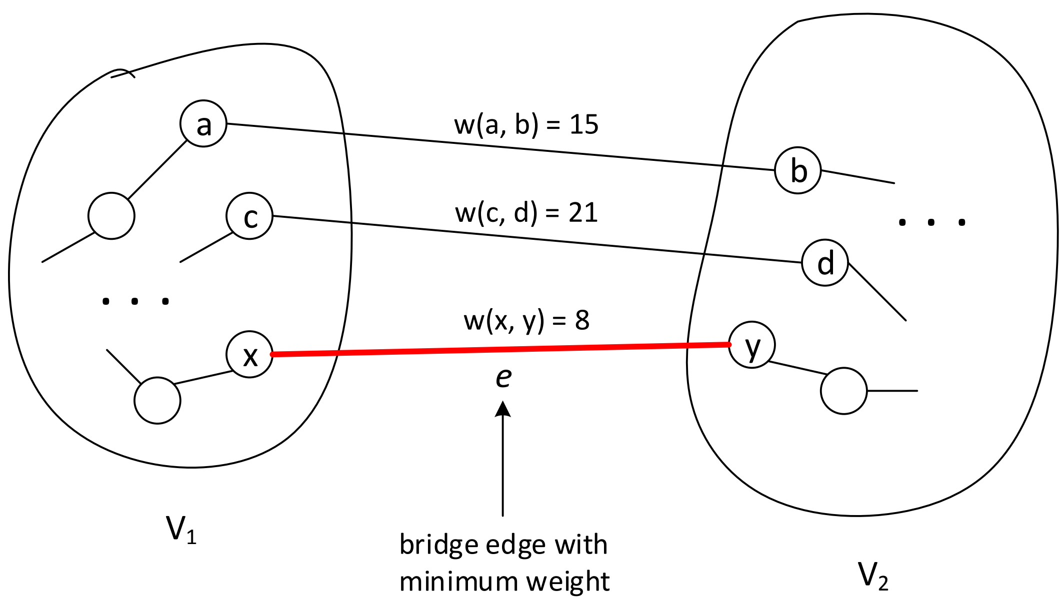
- A *minimum spanning tree* of an undirected, weighted graph *G* is a spanning tree with the minimum weight.
- Minimum spanning tree problem: Find such a tree in *G*.
- Will discuss two algorithms, Prim-Jarnik algorithm and Kruskal's algorithm, both of which are greedy algorithms.

- 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.

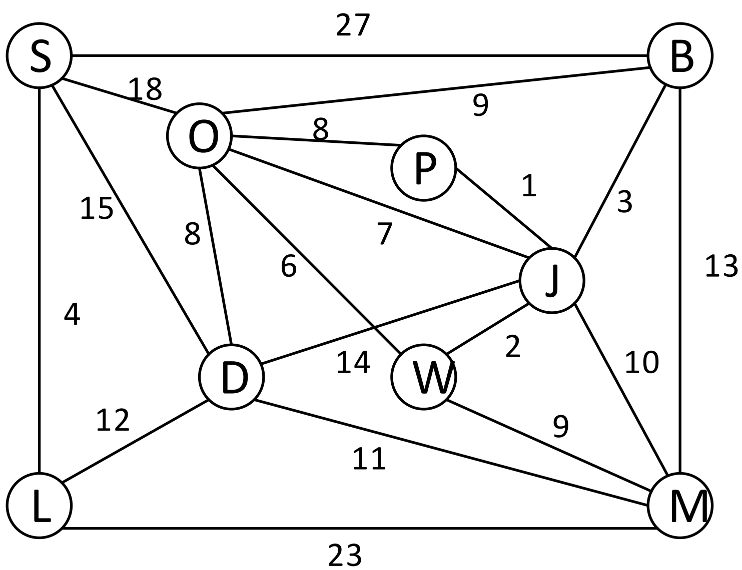
Graph Algorithms
Prim-Jarnik Algorithm

- 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.

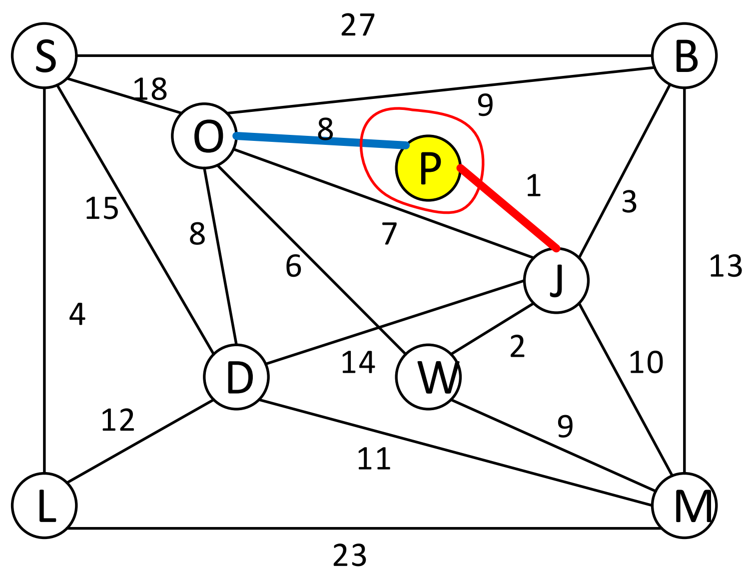
- Pseudocode

Graph Algorithms
Prim-Jarnik Algorithm

- Illustration



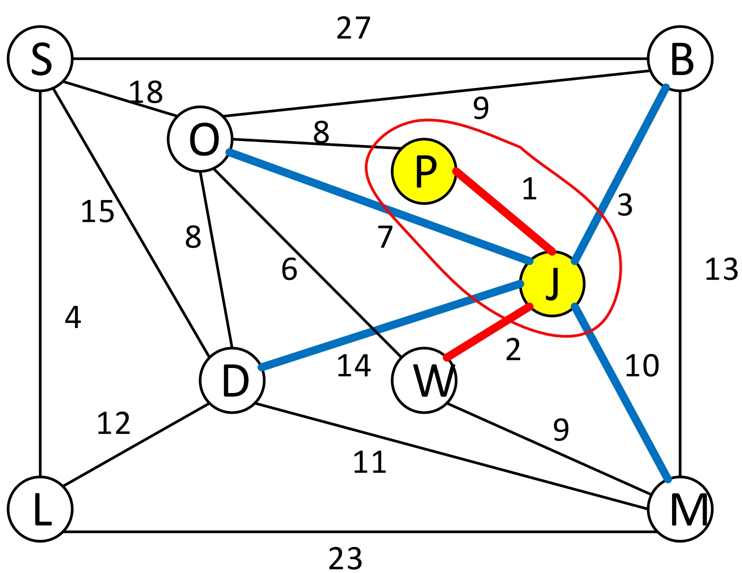
(a) Initial tree



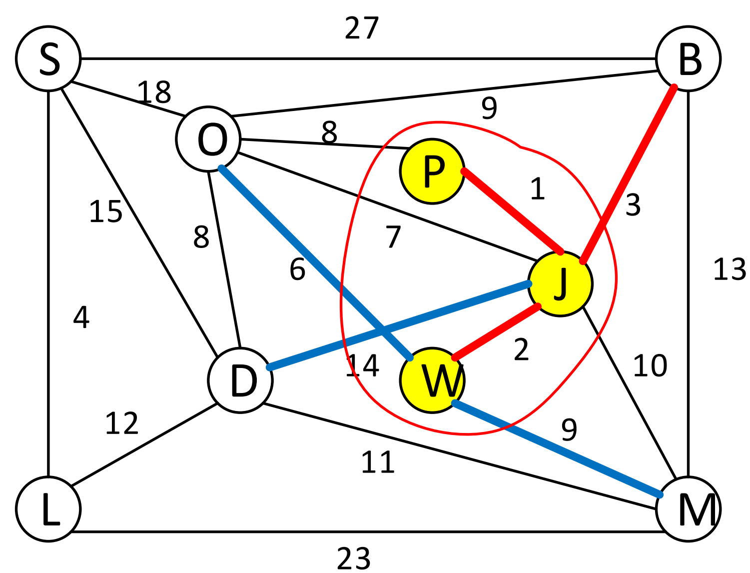
(b) (P,J) is minimum-weight edge.

Graph Algorithms
Prim-Jarnik Algorithm

- Illustration (continued)



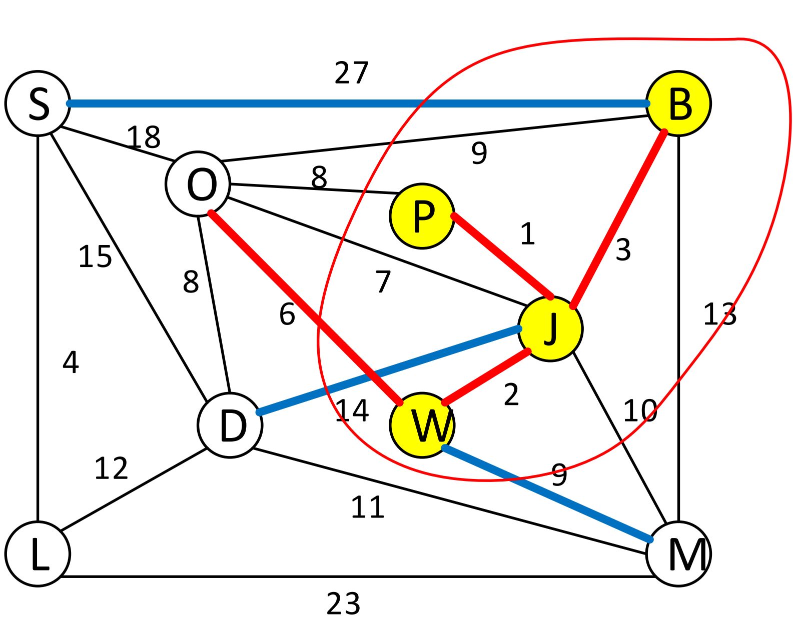
(c) (J,W) is minimum-weight edge.



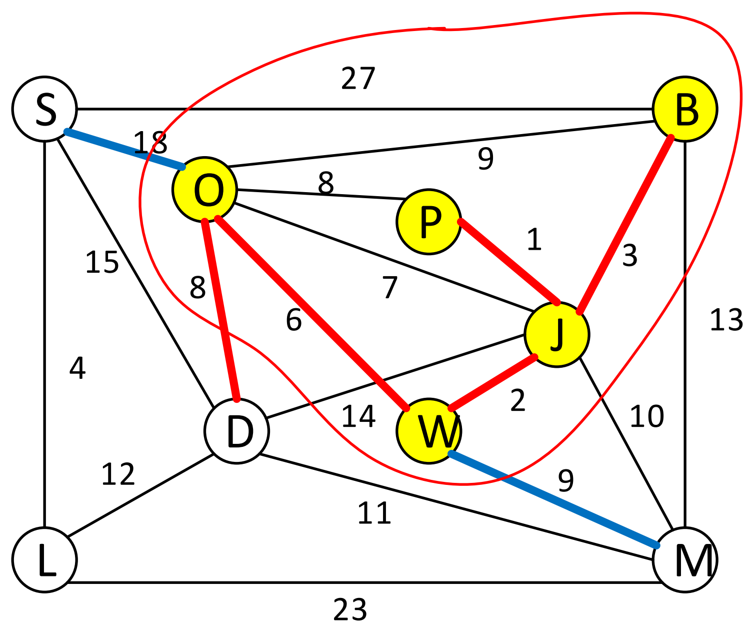
(d) (J,B) is minimum-weight edge.

Graph Algorithms
Prim-Jarnik Algorithm

- Illustration (continued)



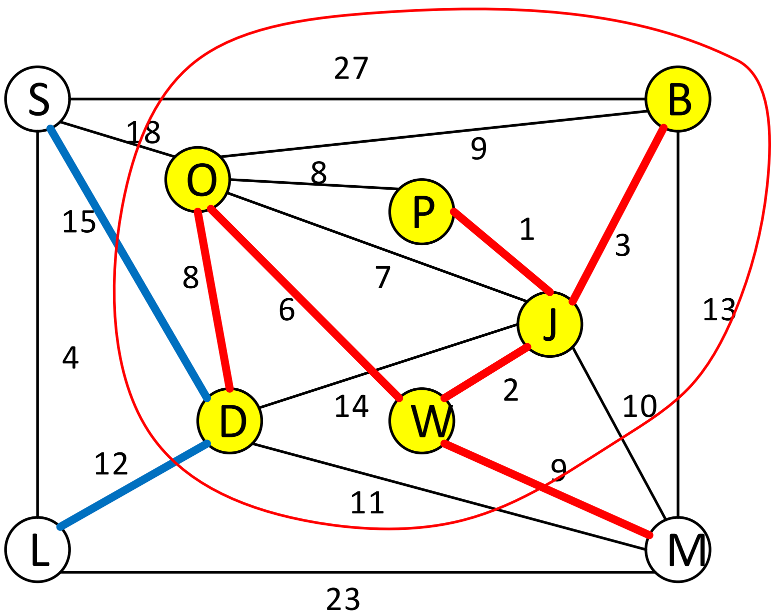
(e) (W,O) is minimum-weight edge.



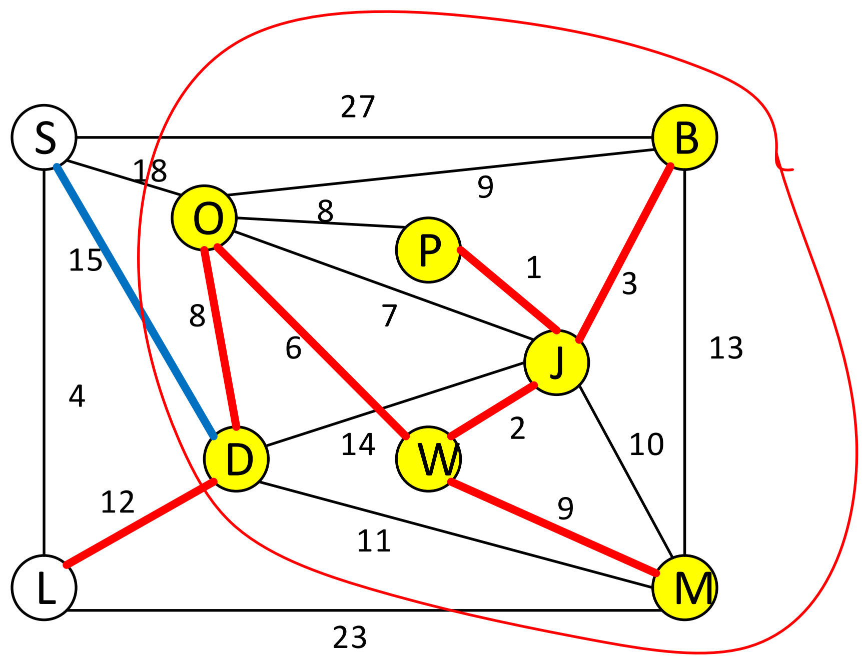
(f) (O,D) is minimum-weight edge.

Graph Algorithms
Prim-Jarnik Algorithm

- Illustration (continued)



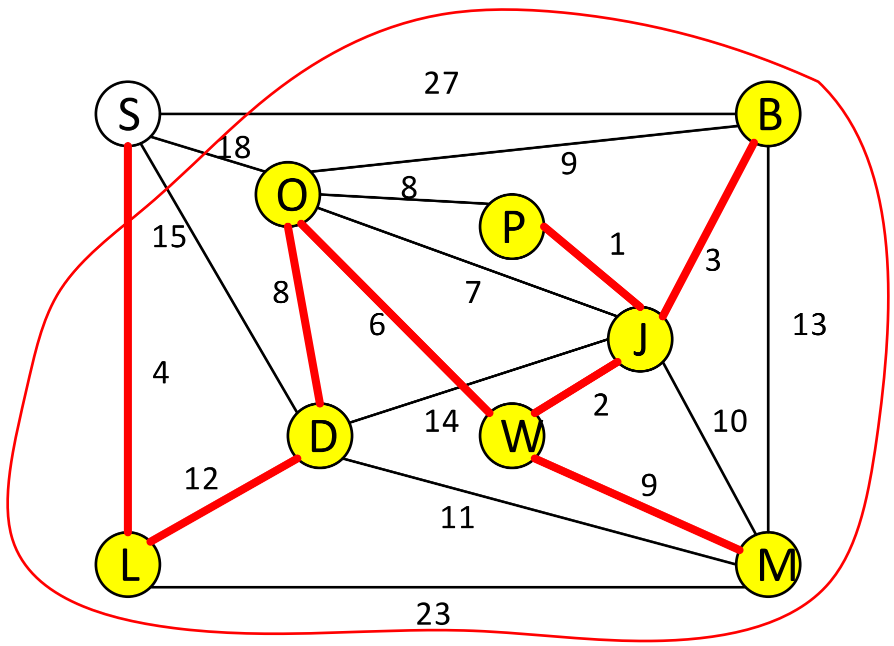
(g) (W,M) is minimum-weight edge.



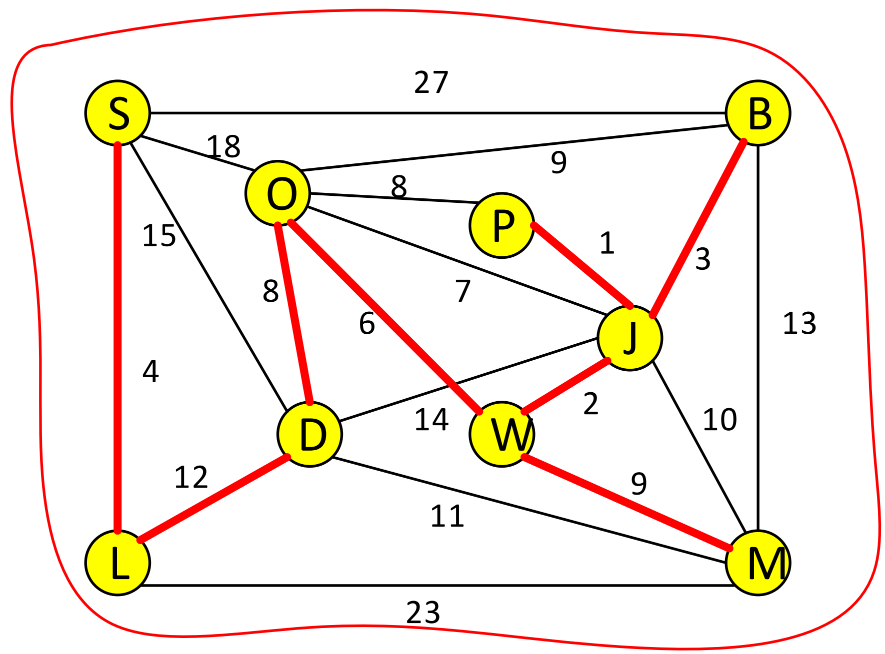
(h) (D,L) is minimum-weight edge.

Graph Algorithms
Prim-Jarnik Algorithm

- Illustration (continued)



(i) (L,S) is minimum-weight edge.



(j) Finished. The thick red edges form a minimum spanning tree *T*.

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

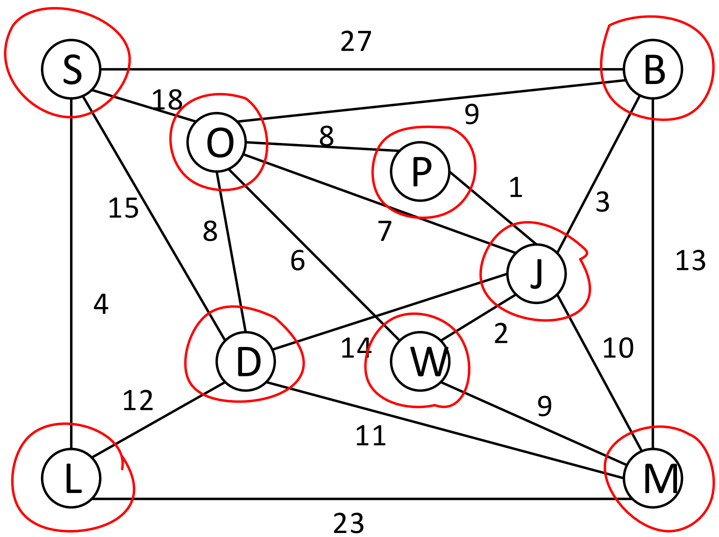
Graph Algorithms
Kruskal's Algorithm

- 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.

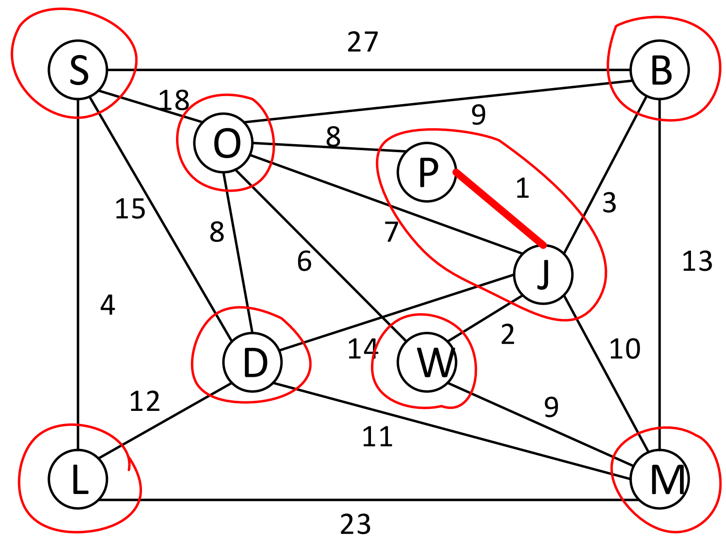
Graph Algorithms
Kruskal's Algorithm

- Pseudocode.

- Illustration



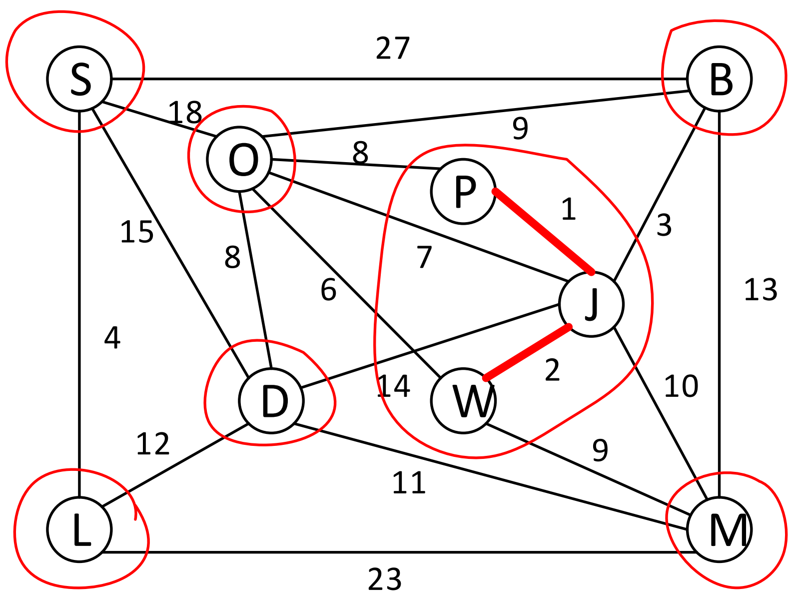
(a) Initial tree. Each vertex is its own cluster. w(J,P) is the smallest.



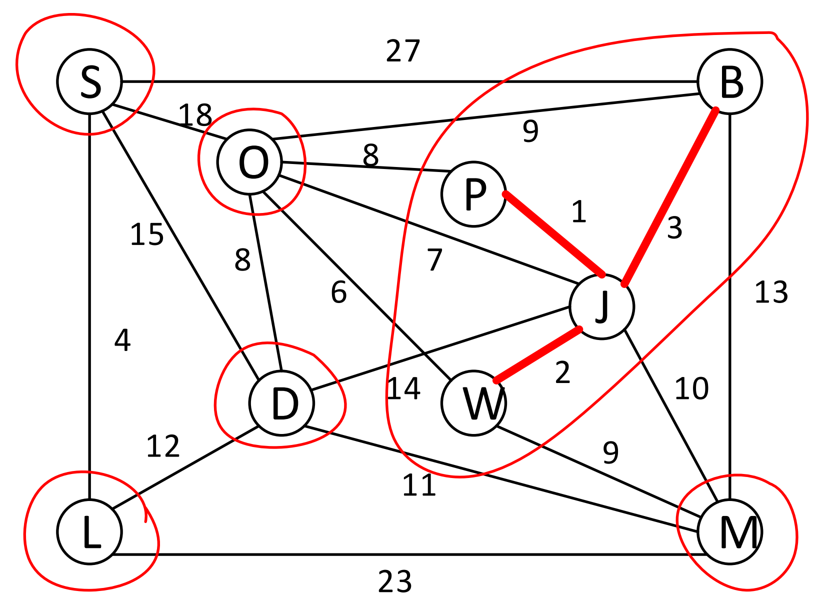
(b) w(J,W) is the next smallest.

Graph Algorithms
Kruskal's Algorithm

- Illustration (continued)



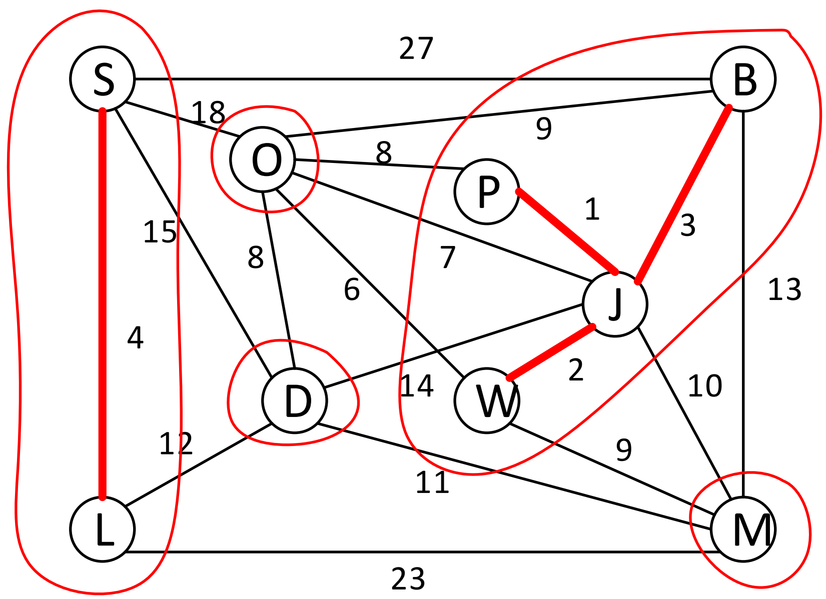
(c) w(B,J) is the next smallest.



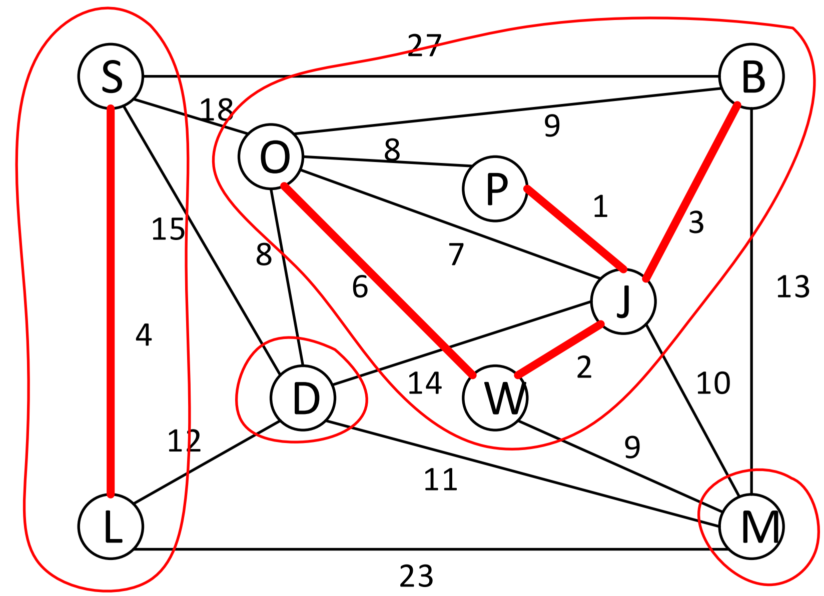
(d) w(L,S) is the next smallest.

Graph Algorithms
Kruskal's Algorithm

- Illustration (continued)



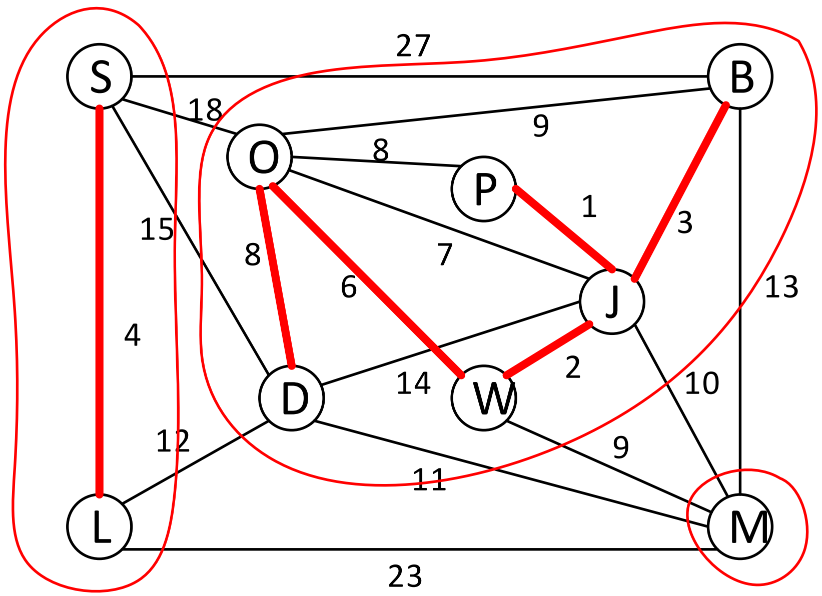
(e) w(O,W) is the next smallest.



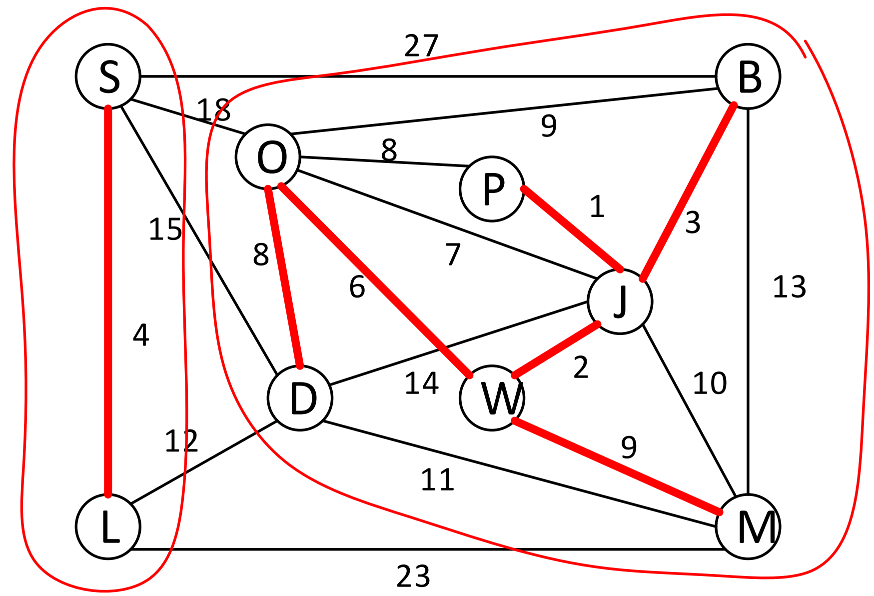
(f) w(J,O) is the next smallest. But, they are in the same cluster. w(O,P) the same. w(D,O) is the next smallest.

Graph Algorithms
Kruskal's Algorithm

- Illustration (continued)



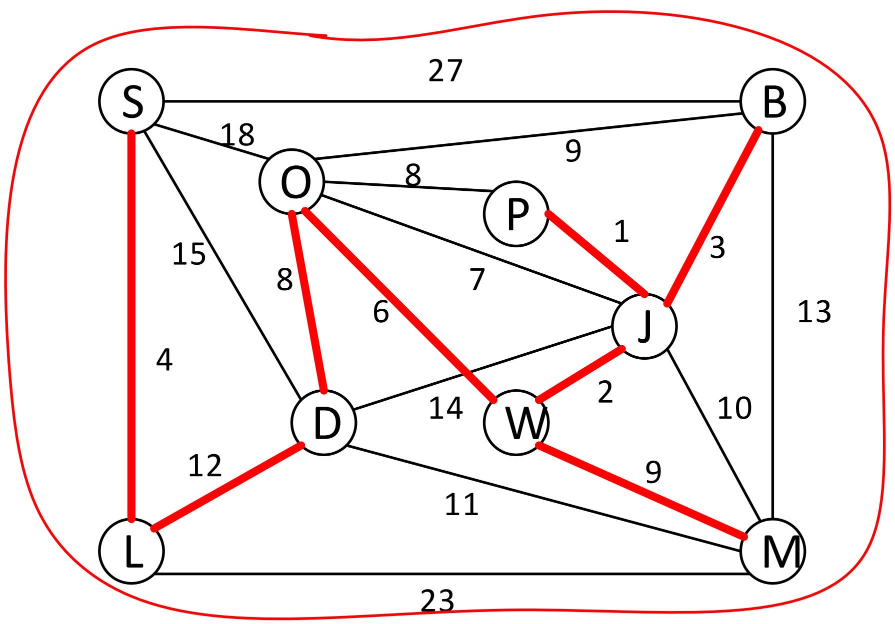
(g) w(B,O) is the next smallest. But, they are in the same cluster. w(M,W) is the next smallest.



(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: $O(m \log n)$

Decision Problem

- Decision problem: A decision problem **P** is a set of questions each of which has a yes or no answer.

- Example: A decision problem $\mathbf{P}_{\mathbf{SQ}}$: Determine whether an arbitrary number is a perfect square or not. This problem consists of the following questions:

  $\mathbf{p}_0$: Is 0 a perfect square?

  $\mathbf{p}_1$: Is 1 a perfect square?

  …

  Here, $\mathbf{p}_i$ is also called an instance of **P.**

Decision Problem

- A solution to a decision problem is an algorithm that determines the answer to every question $\mathbf{p}_i \in \mathbf{P}$.

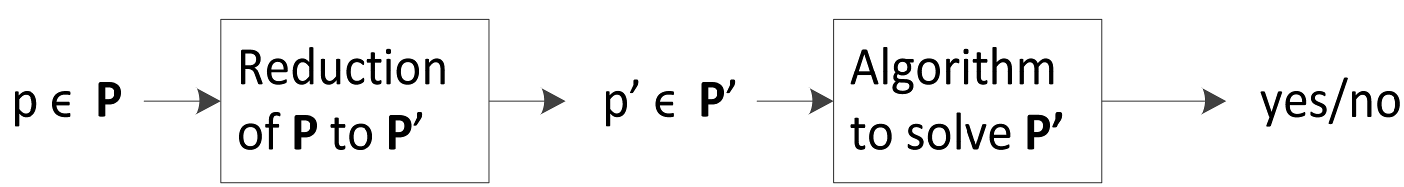
- An algorithm that solves a decision problem should be
  - *complete* – it produces an answer, either positive or negative, to each question in the problem domain
  - *mechanistic* – it consists of a finite sequence of instructions each of which can be carried out without requiring insight, ingenuity, or guesswork
  - *deterministic* – when presented with identical input, it always produces the same result.

Decision Problem

- Decision problems:
  - *Unsolvable* (or *undecidable*)
  - *Solvable*:
    - *Tractable*: A decision problem is said to be tractable if there is at least one polynomially bounded algorithm that solves the problem. Such an algorithm is called an *efficient* algorithm.
    - *Intractable*: A decision problem is said to be intractable if there is no polynomially bounded algorithm (or no efficient algorithm) that solves the problem

Reducibility

- A decision problem **P** is Turing reducible to a problem **P'** if there is a Turing machine that takes any problem $\mathbf{p}_i \in \mathbf{P}$ as input and produces an associated problem $\mathbf{p'}_i \in \mathbf{P'}$ where the answer to the original problem $\mathbf{p}_i$ can be obtained from the answer to $\mathbf{p'}_i$.



P and NP

- A language *L* is decidable in polynomial time if there is a standard (or deterministic) Turing machine *M* that accepts *L* in polynomial time, or $O(n^r)$, where *r* is a natural number independent of *n*.

- The family of languages decidable in polynomial time is denoted ***P*.**

P and NP

- Nondeterministic computation:
  - A deterministic machine solves a decision problem by generating a solution.
  - A nondeterministic machine needs only determine if one of possibilities is a solution.

- A language L is said to be accepted in nondeterministic polynomial time if there is a nondeterministic Turing machine that accepts L in polynomial time, or $O(n^r)$, where r is a natural number independent of n.

P and NP

- The family of languages accepted in nondeterministic polynomial time is denoted ***NP*****.**
- Another definition: A problem is in ***NP*** if it is "verifiable" in polynomial time.
- What "verifiable" means is that given a possible solution (which is also called ***certificate***) we can verify whether it is a solution or not in polynomial time.

P and NP

- ***P = NP ?***

- Unsolved question.
- Since every deterministic machine is also nondeterministic, $\boldsymbol{P} \subseteq \boldsymbol{NP}$**.**
- But it was never proved that $\boldsymbol{NP} \subseteq \boldsymbol{P}$. (If this is proved, then that proves ***P = NP***.)

P and NP

- If Q is reducible to L in polynomial time and $L \in$ ***P*,** then $Q \in$ ***P***.

- A language L is called ***NP-hard*** if for every $Q \in$ ***NP*** Q is reducible to L in polynomial time.
- An ***NP-hard*** language that is also in ***NP*** is called ***NP-complete***.

- If there is an NP-complete language that is also in ***P***, then ***P = NP***.

P and NP

- Two examples of NP-complete problems: Hamiltonian cycle problem and traveling salesman problem.

Hamiltonian Cycle Problem

- A Hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V.
- Note: A cycle is simple if a node, except the first node, is visited only once.
- A graph that contains a Hamiltonian cycle is called "Hamiltonian."
- ***Hamiltonian Cycle Problem***: Does a graph G have a Hamiltonian cycle?

Hamiltonian Cycle Problem

- It can be shown that the Hamiltonian cycle problem can be decidable by a Turing machine in *exponential* time, but not in *polynomial* time. This means Hamiltonian cycle problem is not in ***P***.

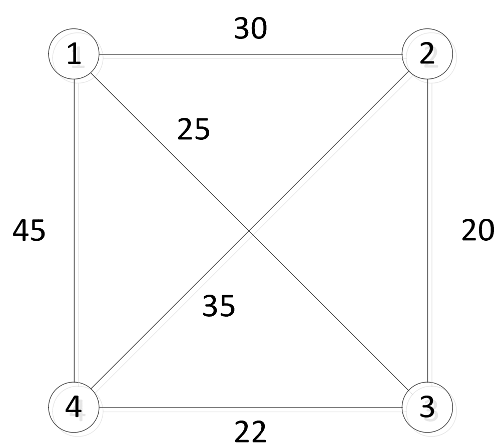
- But, it is decidable in nondeterministic polynomial time.
- Given a cycle in a graph, we can determine whether it is Hamiltonian cycle or not in polynomial time.
- So, Hamiltonian cycle problem is in ***NP***.
- In fact it is an ***NP-complete*** problem.

Traveling Salesman Problem

- Given a complete, non-negative weighted graph, find a Hamiltonian cycle of minimum weight.
- This problem is ***NP-complete***.
- Will briefly discuss three approximate algorithms.

Traveling Salesman Problem

- Consider the following graph:



minimum weight cycle = 1 → 2 → 4 → 3 → 1.

total weight = 30 + 35 + 22 + 25 = 112

Traveling Salesman Problem

- Nearest-neighbor strategy

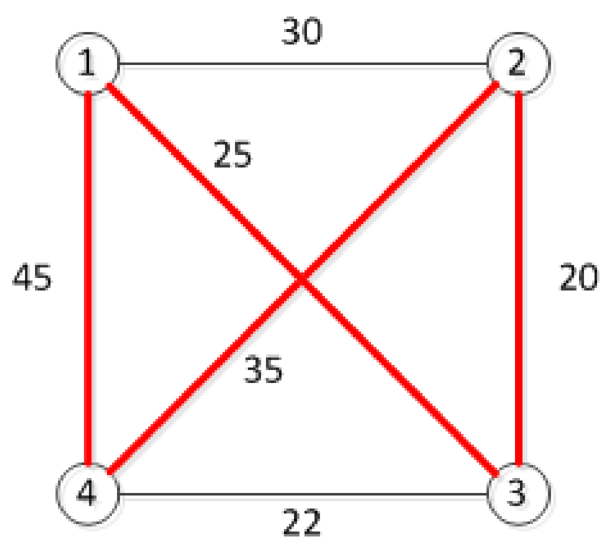
NEAREST-TSP (G, f) /* f is a cost function, or a weight function */

```
select an arbitrary vertex s;
v = s;   Q = {v};   S = G.V – Q;   C = ϕ;
while S != ϕ
      select an edge (v, w) of minimum weight, where w ∈ S;
      C = C ∪ {(v, w)};
      Q = Q ∪ {w};
      S = S – {w};
      v = w;
C = C ∪ {(v, s)};
return C;
```

Running time: $O(V^2)$

Traveling Salesman Problem

- Nearest-neighbor strategy



Starting at vertex 1: (1, 3), (3, 2), (2, 4), (4, 1)

Total weight = 25 + 20 + 35 + 45 = 125

Traveling Salesman Problem

- Shortest-link strategy

SHORTEST-LINK-TSP (G, f)

$R = G.E$;

$C = \phi$;

while R != ϕ

choose the shortest edge (v, w) from R;

$R = R$ – {(v, w)};

if (v, w) does not make a cycle with edges in C and (v, w) would not be the third edge in C incident on v or w

then

$C = C + \{(v, w)\}$;

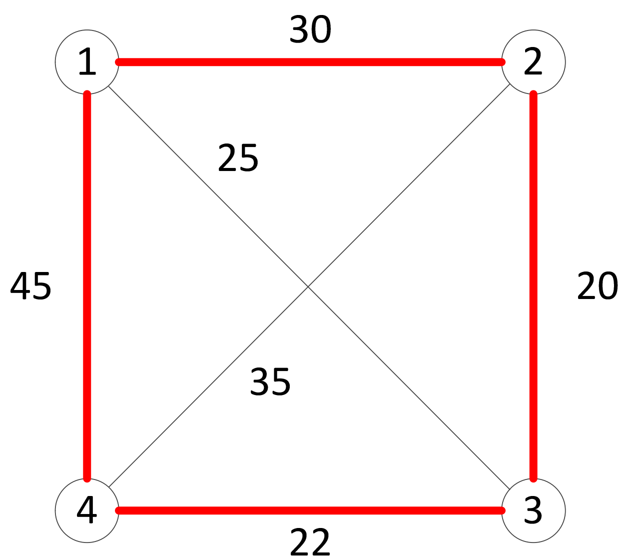
add the edge connecting the end points of the path in C;

return C;

Running time: $O(E \log V)$

Traveling Salesman Problem

- Shortest-link strategy



Edges added: (2, 3), (3, 4), (2, 1), (1, 4)

Total weight = 20 + 22 + 30 + 45 = 117

Traveling Salesman Problem

- In general, we cannot establish a bound on how much the weight of an approximate algorithm differ from the weight of a minimum tour.

- If we assume the triangle inequality holds on distances among vertices, we can develop an approximate algorithm that has an upper bound on the weight.

- Triangle inequality:

  $f(u, v) \leq f(u, w) + f(w, v)$, for all $u, v, w \in G.V$.

- Euclidean distance has the triangle inequality property.

Traveling Salesman Problem

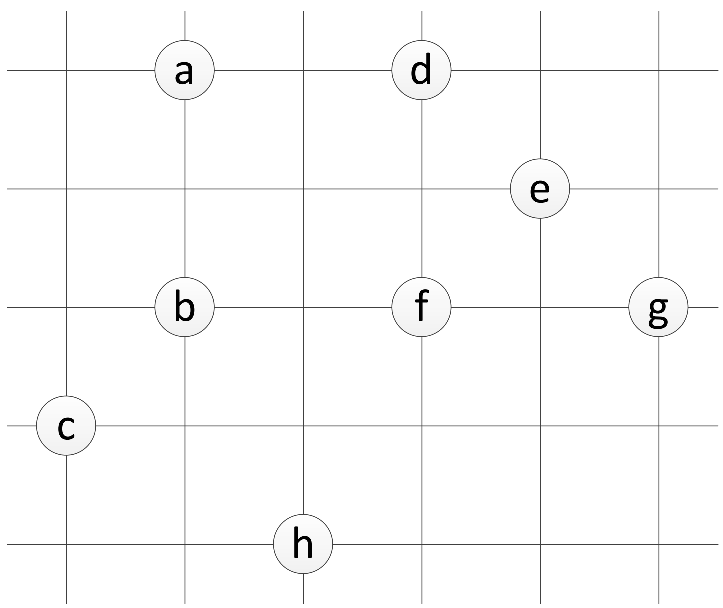
- The following approximate algorithm has an upper bound on the weight: total weight of a cycle is no more than the twice that of the minimum spanning tree's weight

APPROX-TSP-TOUR (G, f)

select a vertex $r \in G.V$ to be the root;
compute MST T from r using MST-PRIM(G, f, r);
let H be a list of vertices, ordered according to when they are
    first visited in a preorder tree walk of T;
return H

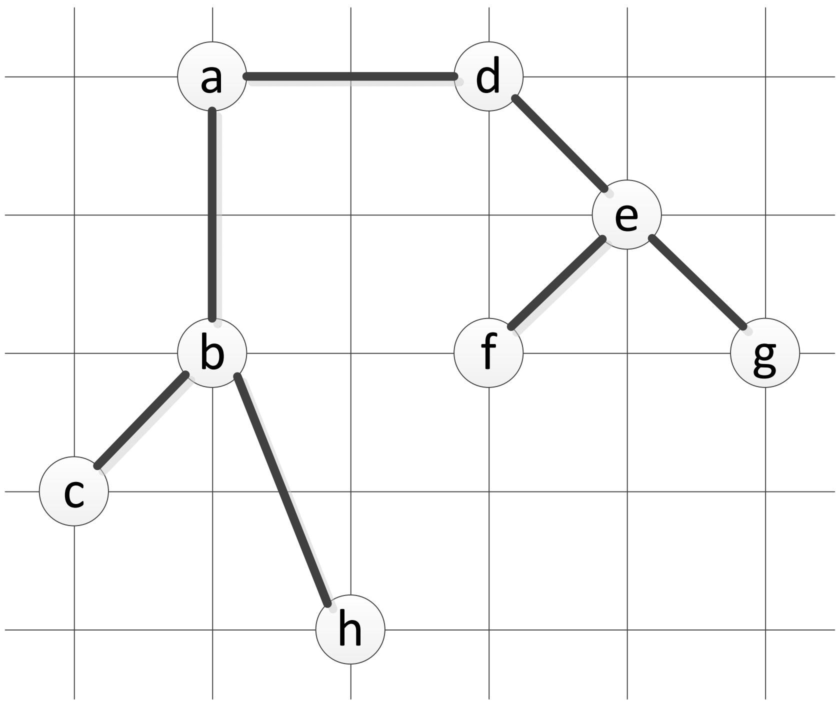
Traveling Salesman Problem

- Example: Given the following complete graph (There are edges from each node to all other nodes though edges are not shown in the graph below).



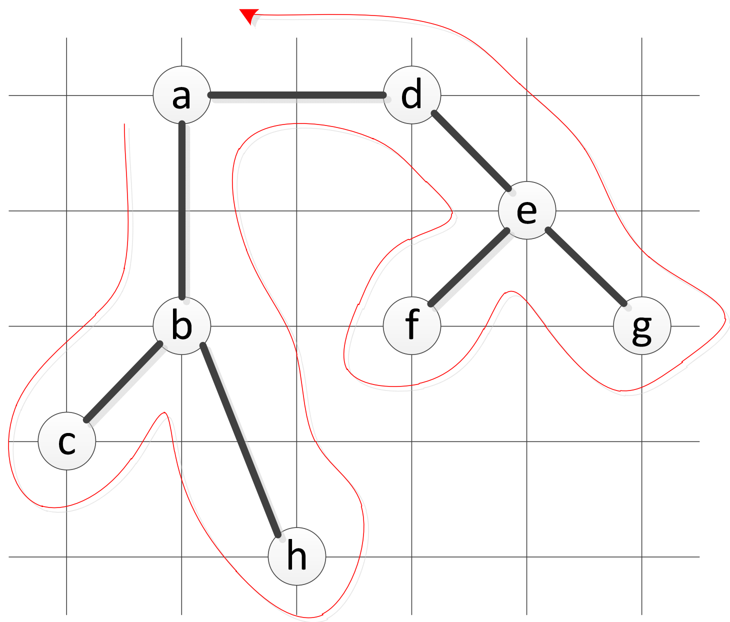
Traveling Salesman Problem

- A minimum spanning tree *T* (*a* is the root)



Traveling Salesman Problem

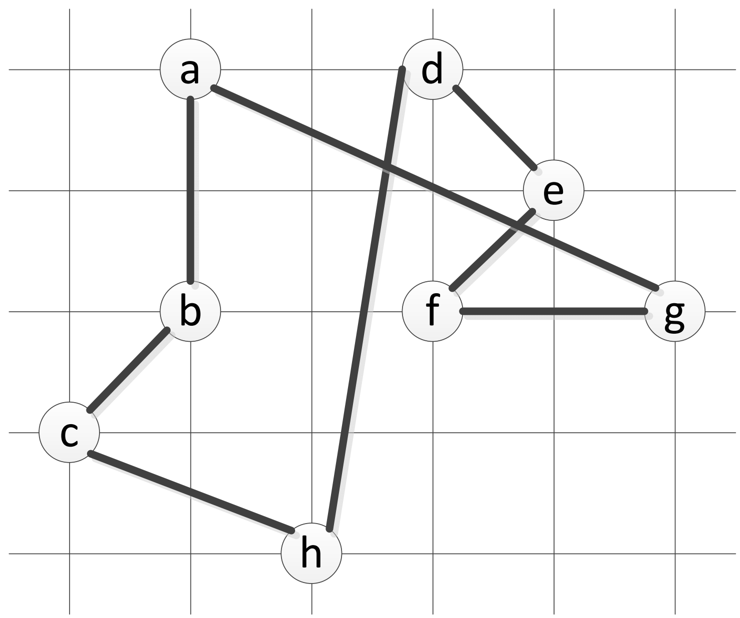
- A minimum spanning tree T (a is the root)



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

Traveling Salesman Problem

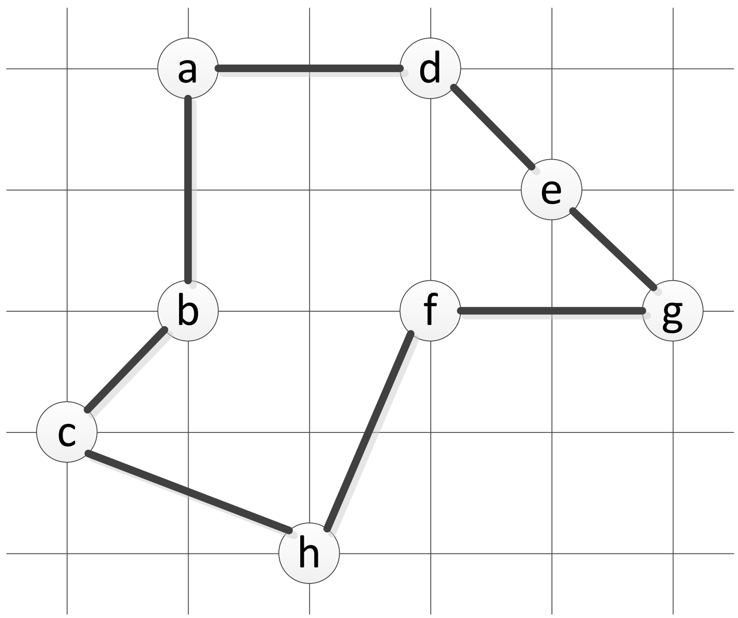
- *H* returned by APPROX-TSP-TOUR is



total weight = approx. 19.074

Traveling Salesman Problem

- An optimal tour (or Hamiltonian cycle with minimum weight)



total weight = approx. 14.715

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.
- T.A. Sudkamp, “Languages and Machines,” 1988, Addison Wesley.
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, “Introduction to Algorithms,” 3rd Ed., 2009, MIT Press.