University of Science HCMC
Faculty of Information Technology

Data Structure and Algorithms Course

Graph

Lecturer: Le Ngoc Thanh
Email: lnthanh@fit.hcmus.edu.vn

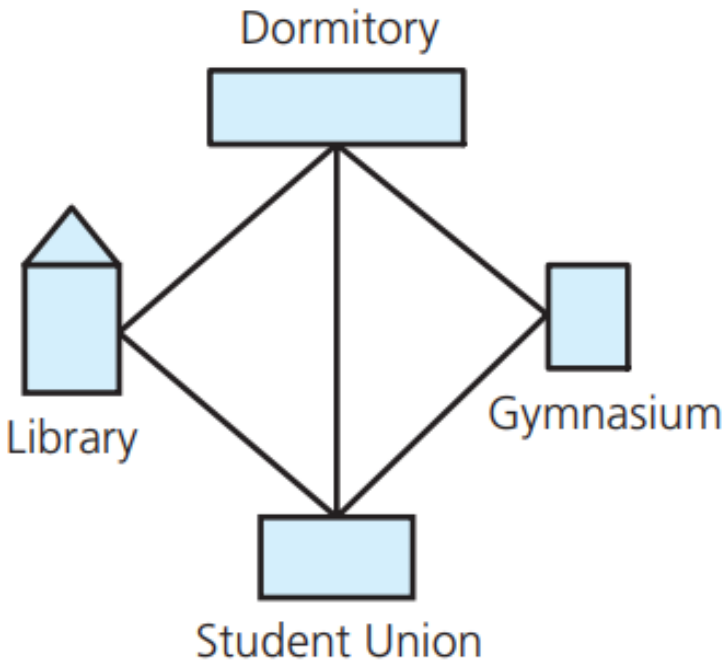
HCM City

Content

- Introduction
- Basic Concepts
- Graph Abstract Data Type
- Graph Traversal
- Graph Applications
- Some Difficult Problems

Graph

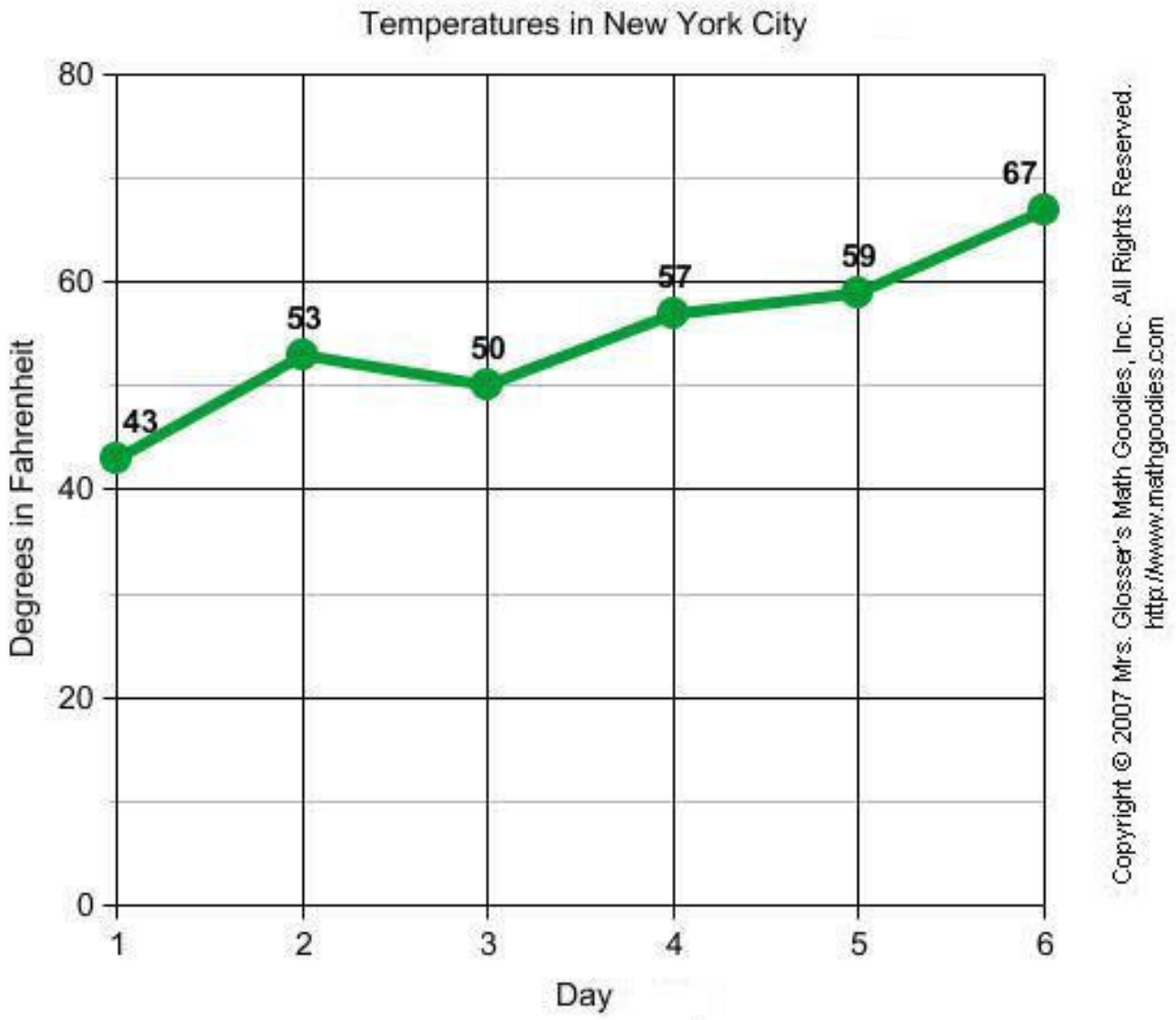
- A **graph** *G* consists of two sets: a set *V* of vertices, or nodes, and a set *E* of edges that connect the vertices.



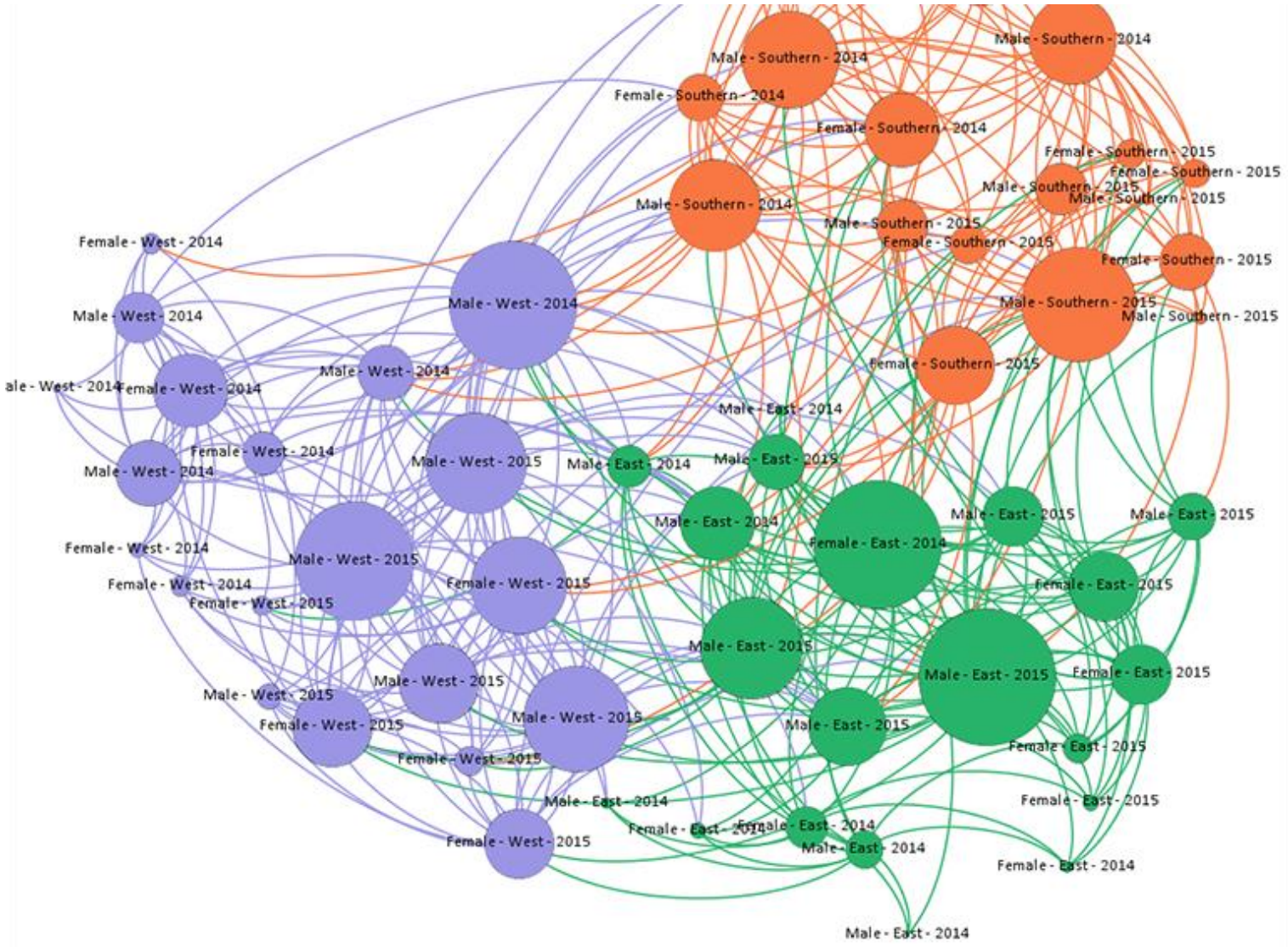
For example, the campus map is a graph whose vertices represent buildings and whose edges represent the sidewalks between the buildings.

Examples – US map

Examples – Temperature chart



Example – social network



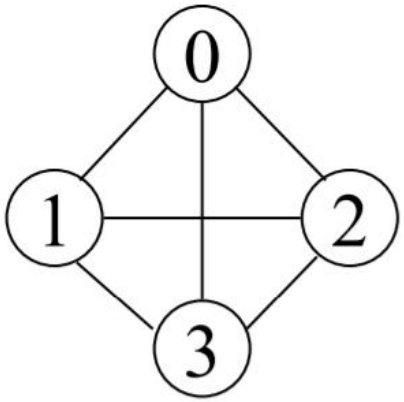
Content

- Introduction
- Basic Concepts
- Graph Abstract Data Type
- Graph Traversal
- Graph Applications
- Some Difficult Problems

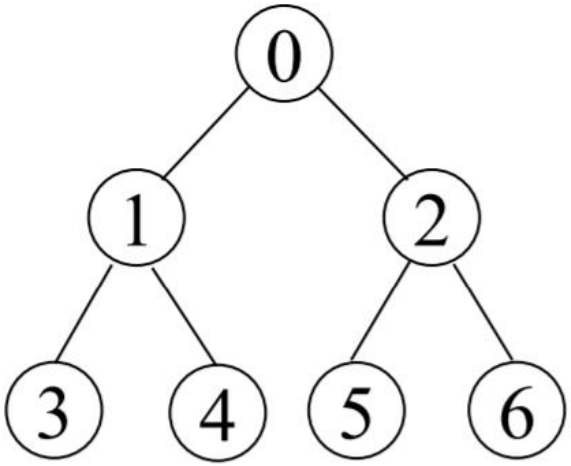
Formal Definition

- A graph, *G=(V, E)*, consists of two sets:
  - a finite non empty set of vertices(V), and
  - a finite set (E) of unordered pairs of distinct vertices called edges.
  - V(G) and E(G) represent the sets of vertices and edges of G, respectively.
- In graph theory, a vertex or node or points is the fundamental unit out of which graphs are formed.
- Edge or Arcs or Links: Gives the relationship between the two vertices.

Examples for Graph



G_1



G_2

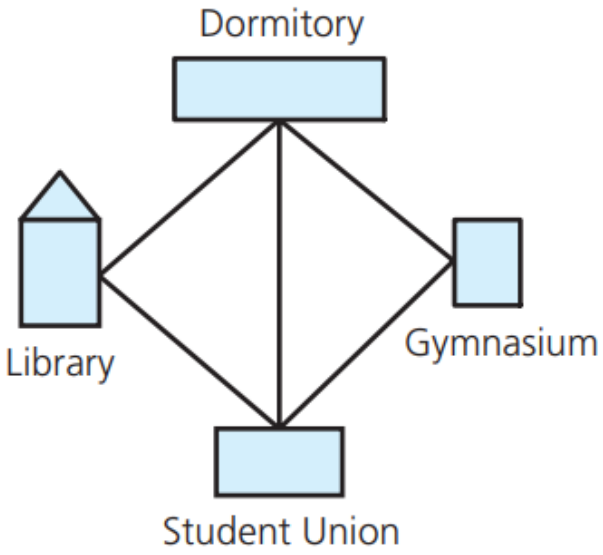
G_3

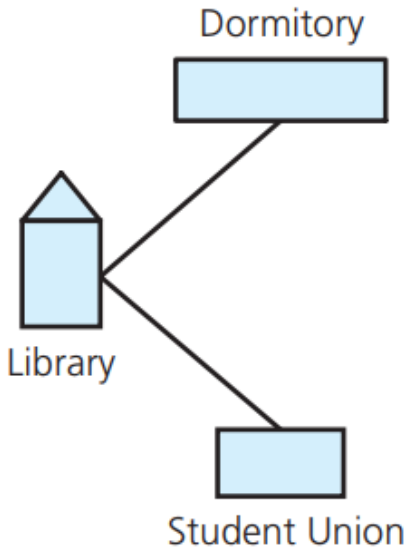
$V(G_1)=\{0,1,2,3\}$
$V(G_2)=\{0,1,2,3,4,5,6\}$
$V(G_3)=\{0,1,2\}$

$E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

Subgraph

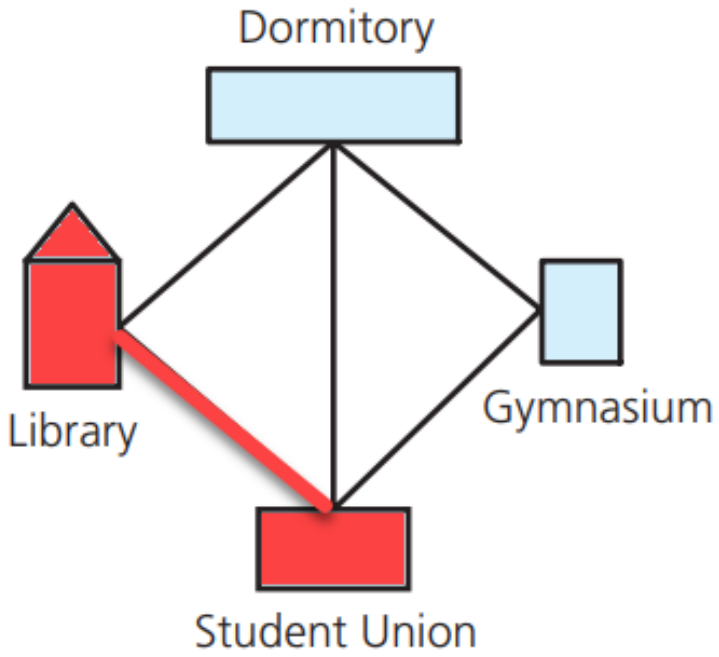
- A **subgraph** consists of a subset of a graph's vertices and a subset of its edges.





Adjacent

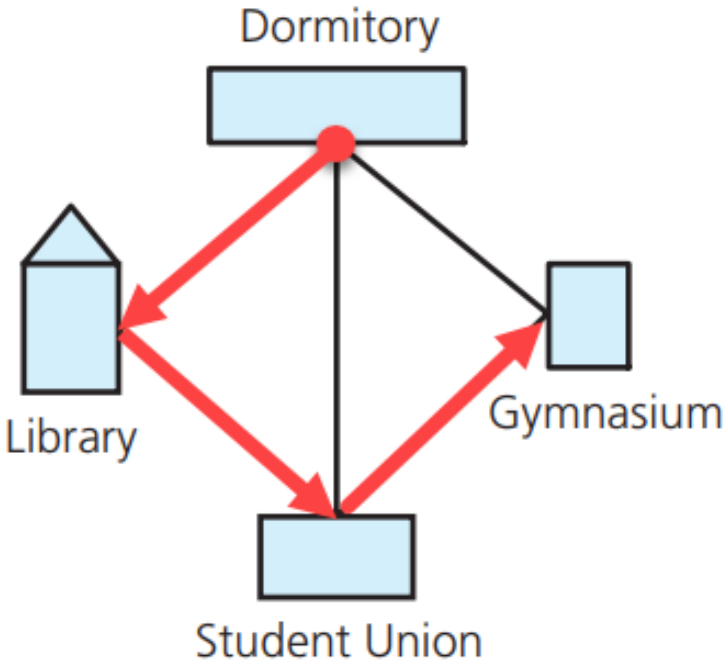
- Two vertices of a graph are **adjacent** if they are joined by an edge.



The Library and the Student Union are *adjacent*.

Path

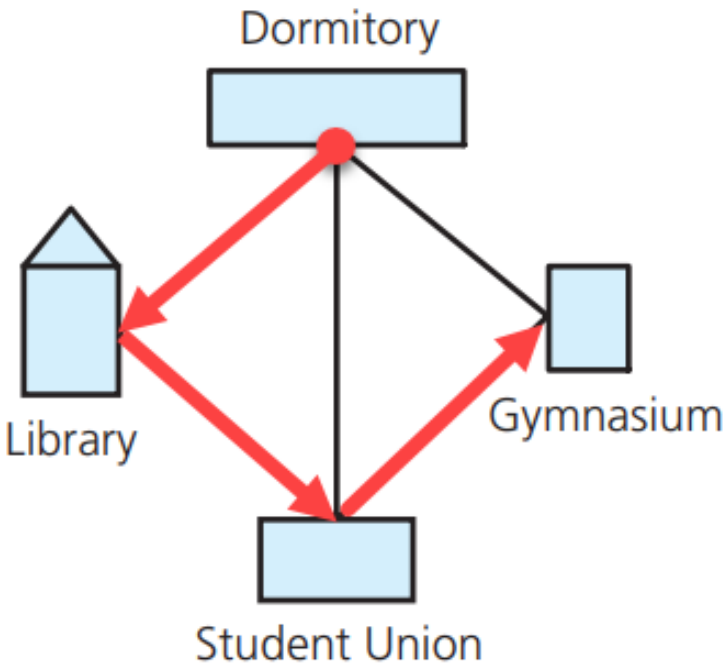
- A **path** between two vertices is a sequence of edges that begins at one vertex and ends at another vertex.



There is a *path* that begins at the Dormitory, leads first to the Library, then to the Student Union, and finally to the Gymnasium

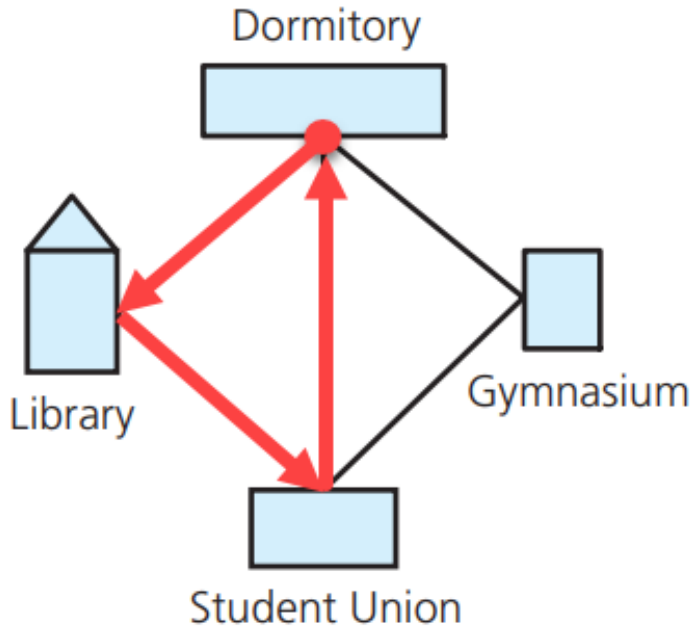
Simple Path

- A **simple path** is a path does not pass through the same vertex more than once.



Cycle

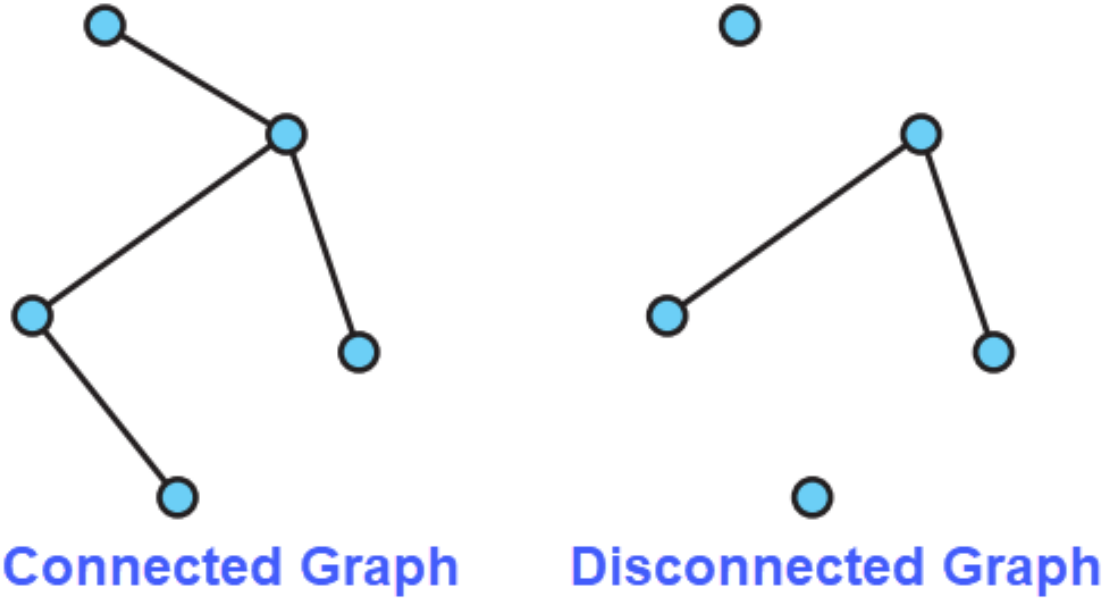
- A **cycle** is a path that begins and ends at the same vertex; a **simple cycle** is a cycle that does not pass through other vertices more than once.



The path Library–Student Union–Gymnasium–Dormitory–Library is a *simple cycle* in the graph

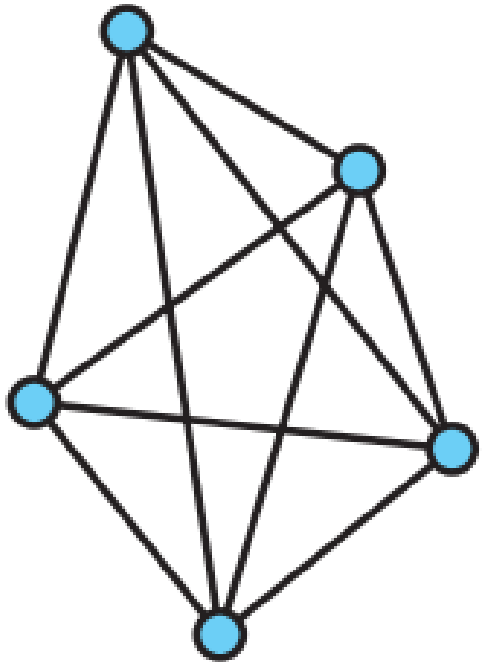
Connected vs Disconnected Graph

- A graph is **connected** if each pair of distinct vertices has a path between them.
  - That is, in a connected graph you can get from any vertex to any other vertex by following a path.
- Otherwise, a **disconnected** graph does not necessarily.



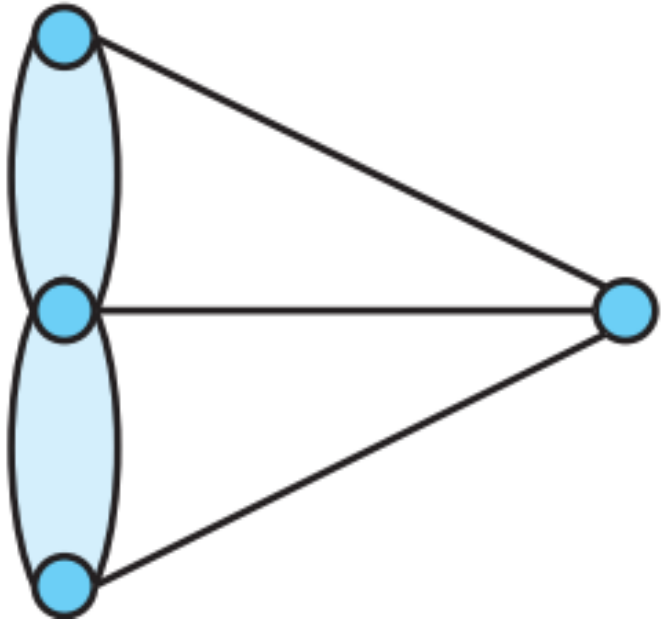
Complete graph

- In a **complete graph**, each pair of distinct vertices has an edge between them.
  - Clearly, a complete graph is also *connected*, but the *converse is not true*.



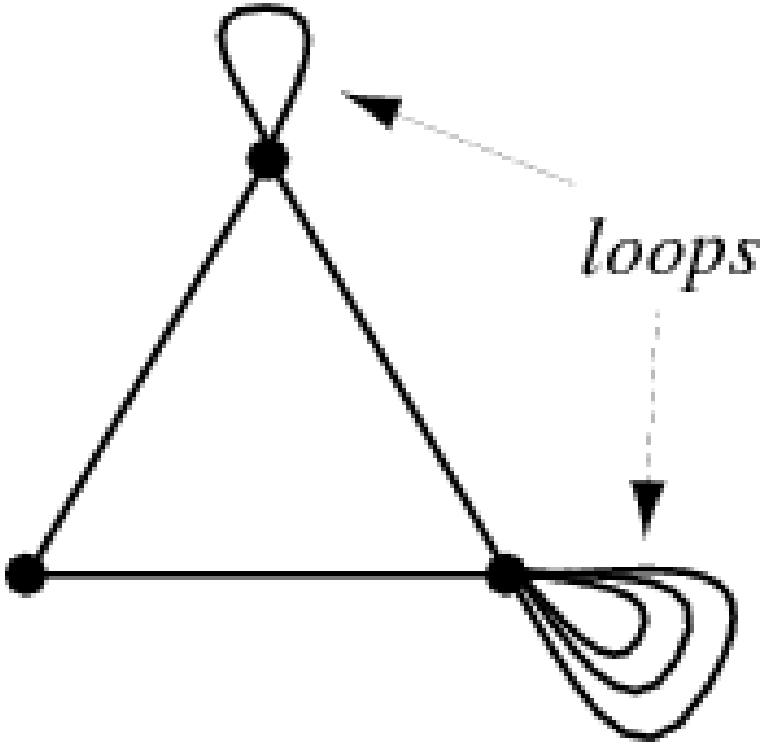
Multigraph

- Because a *graph* has a set of edges, a graph *cannot have duplicate edges between vertices*.
- However, a **multigraph** does allow multiple edges.
  - Thus, *a multigraph is not a graph*. A graph's edges cannot begin and end at the same vertex.



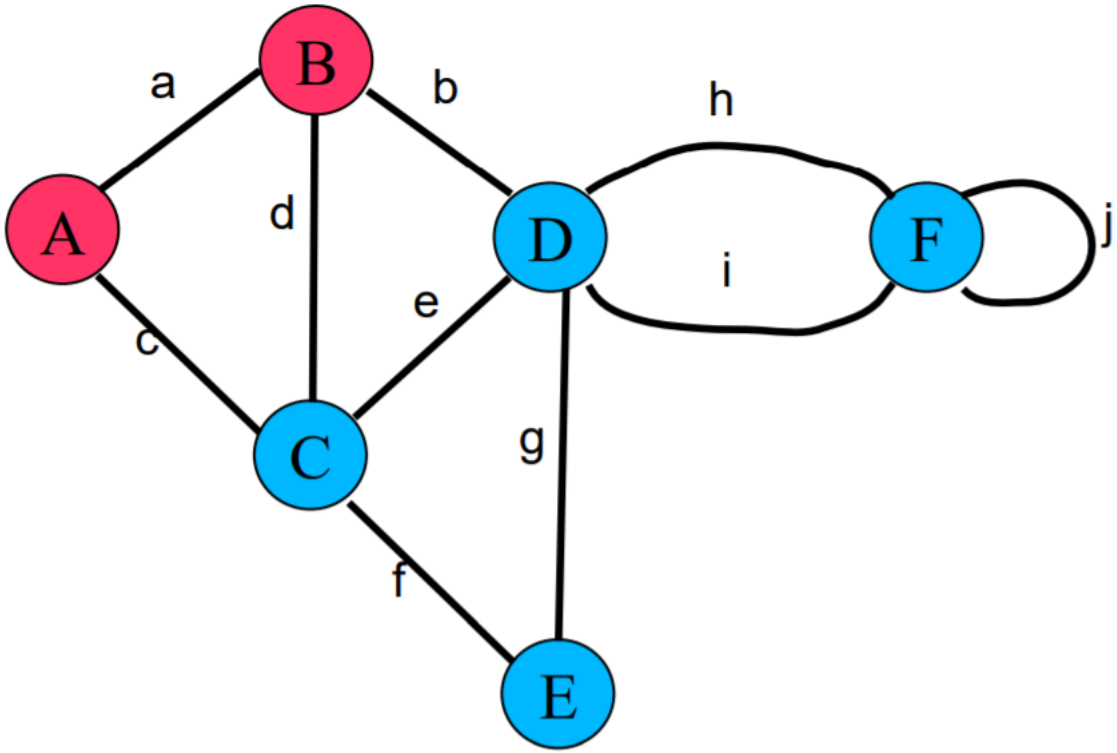
Self edge

- A graph's edges begin and end at the same vertex is called a **self edge** or **loop**.



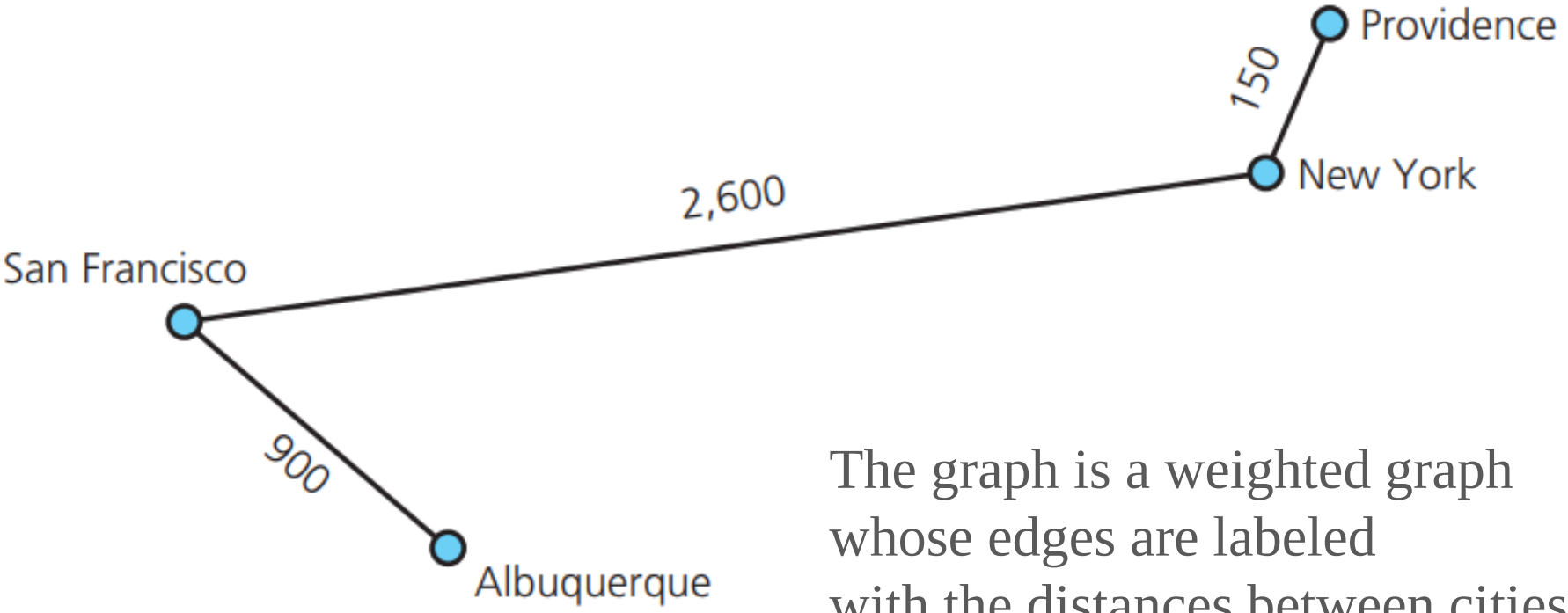
Label

- You can *label* the edges of a graph.



Weighted graph

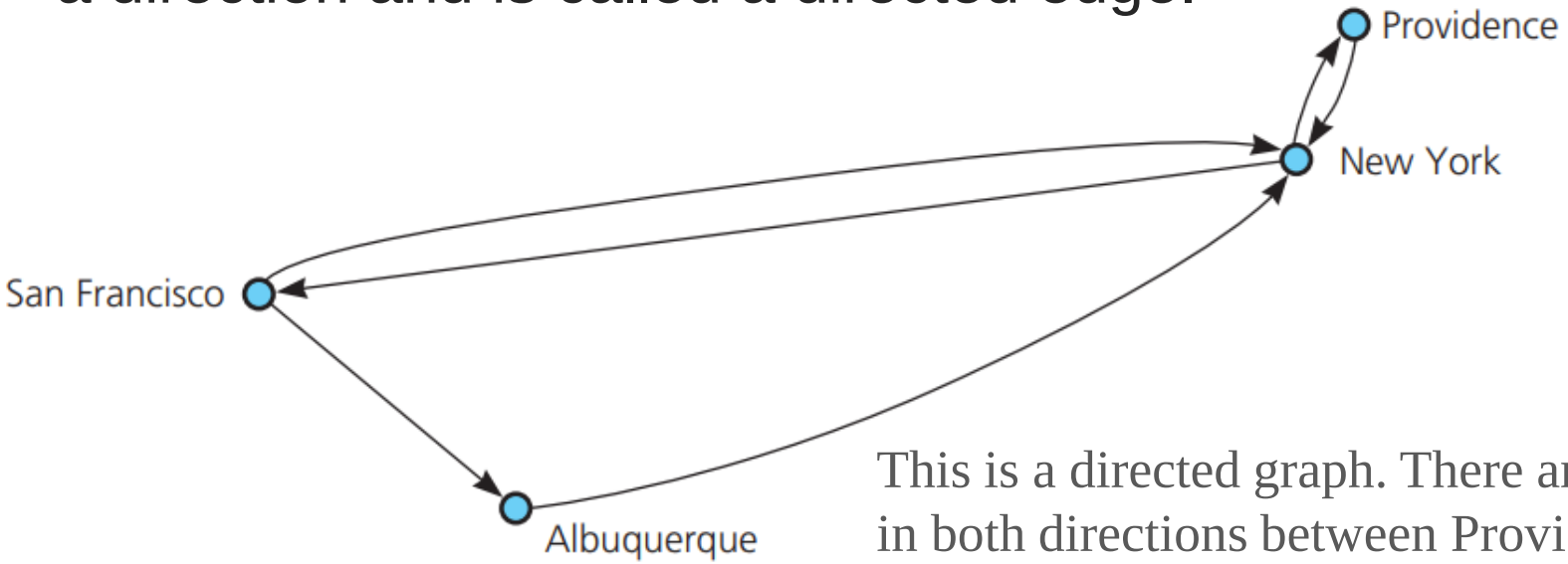
- When these labels represent *numeric values*, the graph is called a **weighted graph**.



The graph is a weighted graph whose edges are labeled with the distances between cities

Undirected and directed graphs

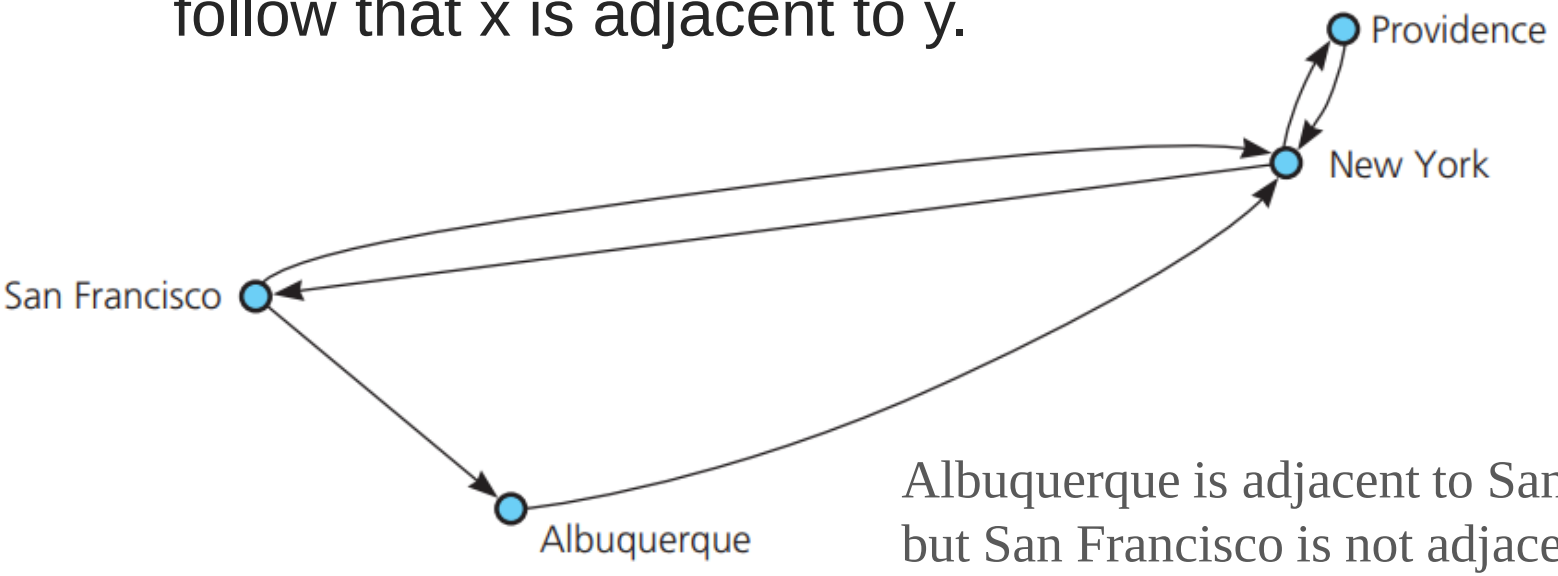
- In **undirected graphs**, the edges do not indicate a direction. That is, you can travel in either direction along the edges between the vertices of an undirected graph.
- In contrast, each edge in a **directed graph**, or digraph, has a direction and is called a directed edge.



This is a directed graph. There are flights in both directions between Providence and New York, but, although there is a flight from San Francisco to Albuquerque, there is no flight from Albuquerque to San Francisco.

Undirected and directed graphs

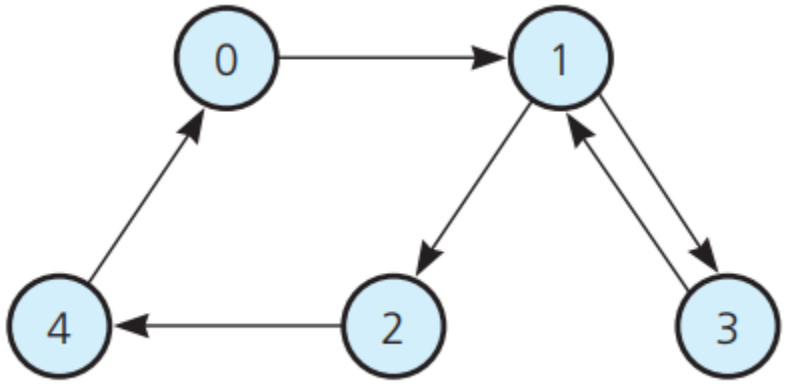
- The definition of adjacent vertices is not quite same in both graph type.
- In digraph:
  - If there is a directed edge from vertex x to vertex y, then y is adjacent to x. (Alternatively, y is a *successor* of x, and x is a *predecessor* of y.) It does not necessarily follow that x is adjacent to y.



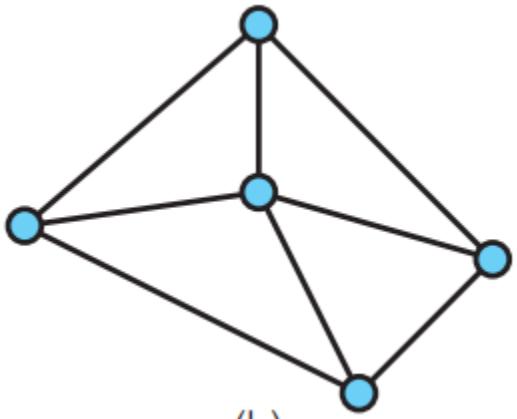
Albuquerque is adjacent to San Francisco, but San Francisco is not adjacent to Albuquerque.

Exercises

- Describe the following graphs. For example, are they directed? Connected? Complete? Weighted?



(a)


(b)

Content

- Introduction
- Basic Concepts
- Graph Abstract Data Type
- Graph Traversal
- Graph Applications
- Some Difficult Problems

Graph as ADT

- Graphs as abstract data types with operations:
  - Test whether a graph is empty.
  - Get the number of vertices in a graph.
  - Get the number of edges in a graph.
  - See whether an edge exists between two given vertices.
  - Insert a vertex in a graph whose vertices have distinct values that differ from the new vertex's value.
  - Insert an edge between two given vertices in a graph.
  - Remove a particular vertex from a graph and any edges between the vertex and other vertices.
  - Remove the edge between two given vertices in a graph.
  - Retrieve from a graph the vertex that contains a given value

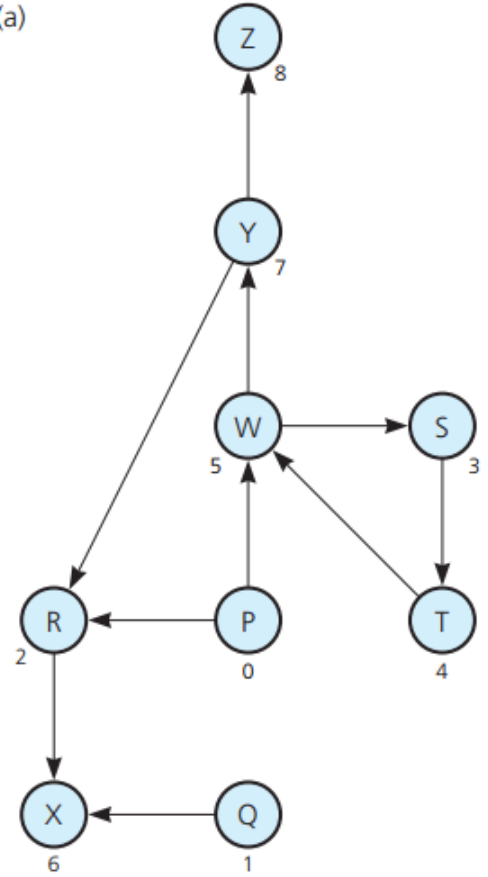
Store graphs in a computer system

- There are different ways to store graphs in a computer system.
- The data structure used depends on both the graph structure and the algorithm used for manipulating the graph.
- The two most common implementations of a graph are the adjacency matrix and the adjacency list.

Adjacency matrix

- An adjacency matrix for a graph with n vertices numbered 0, 1, . . ., n – 1 is:
  - an n by n array matrix such that matrix[i][j] is 1 (true) if there is an edge from vertex i to vertex j, and 0 (false) otherwise.

(a)



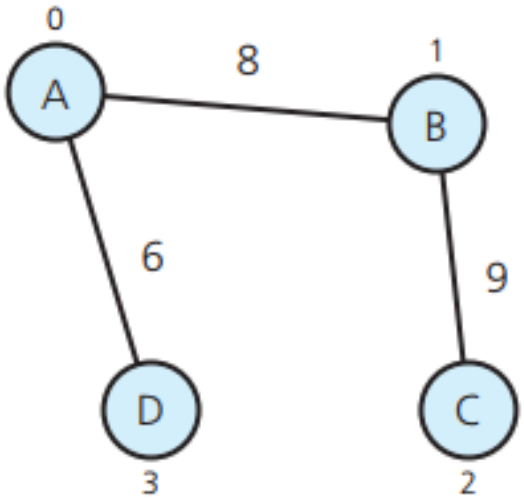
(b)

		0 P	1 Q	2 R	3 S	4 T	5 W	6 X	7 Y	8 Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Adjacency matrix

- When the graph is weighted:
  - let matrix[i][j] be the weight that labels the edge from vertex *i* to vertex *j*, instead of simply 1
  - let matrix[i][j] equal ∞ instead of 0 when there is no edge from vertex *i* to vertex *j*.

(a)



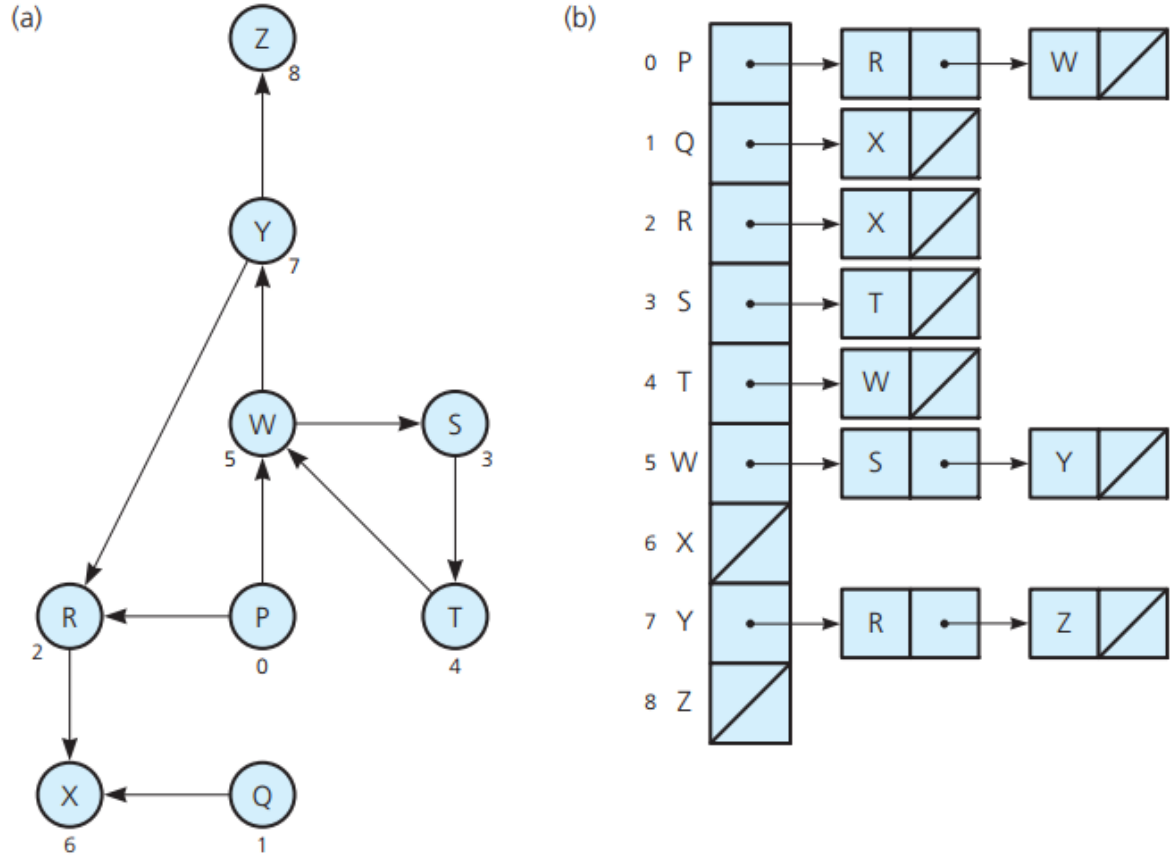
(b)

		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Notice that the adjacency matrix for an undirected graph is symmetrical; that is, matrix[i][j] equals matrix[j][i].

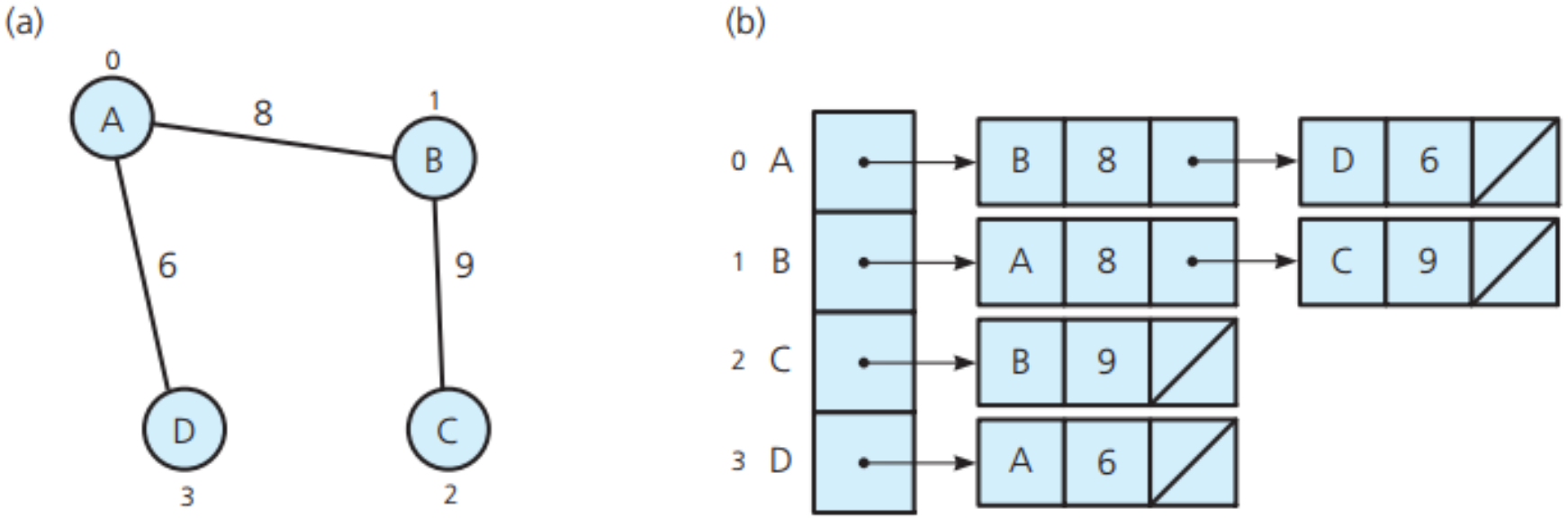
Adjacency list

- An adjacency list for a graph with n vertices numbered 0, 1, . . ., n – 1 consists of n linked chains.
  - The i^{th} linked chain has a node for vertex j if and only if the graph contains an edge from vertex i to vertex j. This node can contain the vertex j's value, if any.



Adjacency list

- The adjacency list for an undirected graph treats each edge as if it were two directed edges in opposite directions.



Which is better?

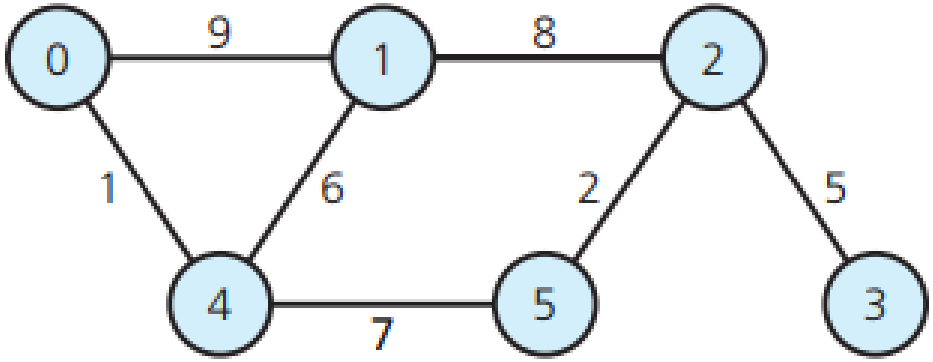
- Which of these two implementations of a graph—the adjacency matrix or the adjacency list—is better?
  - The answer depends on how your particular application uses the graph.
- For example, the two most commonly performed graph operations are:
  - Determine whether there is an edge from vertex i to vertex j → adjacency matrix
  - Find all vertices adjacent to a given vertex I → adjacency list

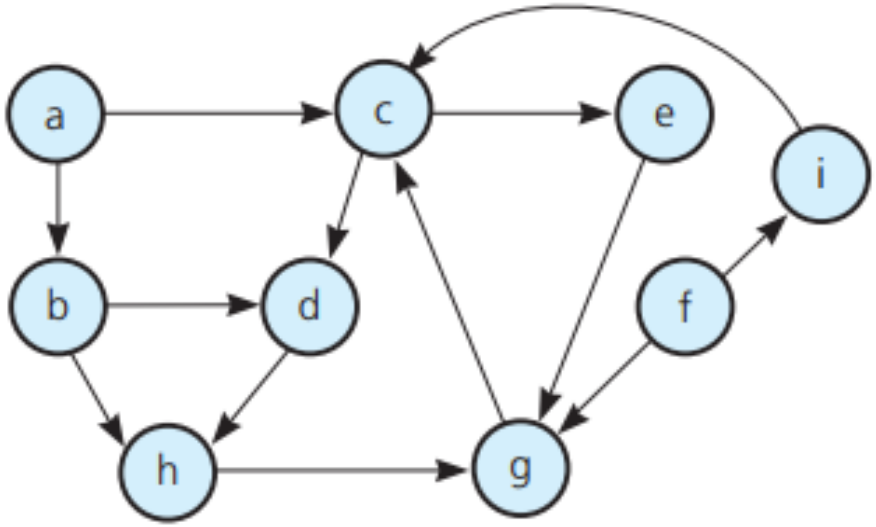
Which is better?

- Consider the space requirements of the two implementations.
  - On the surface, the adjacency matrix requires less memory than the adjacency list, because each entry in the matrix is simply an integer, whereas each list node contains both a value to identify the vertex and a pointer.
  - The adjacency matrix, however, always has n^2 entries, whereas the number of nodes in an adjacency list equals the number of edges in a directed graph or twice that number for an undirected graph. Even though the adjacency list also has n head pointers, it often requires less storage than an adjacency matrix

Exercies

- Give the adjacency matrix and adjacency list for two graph:



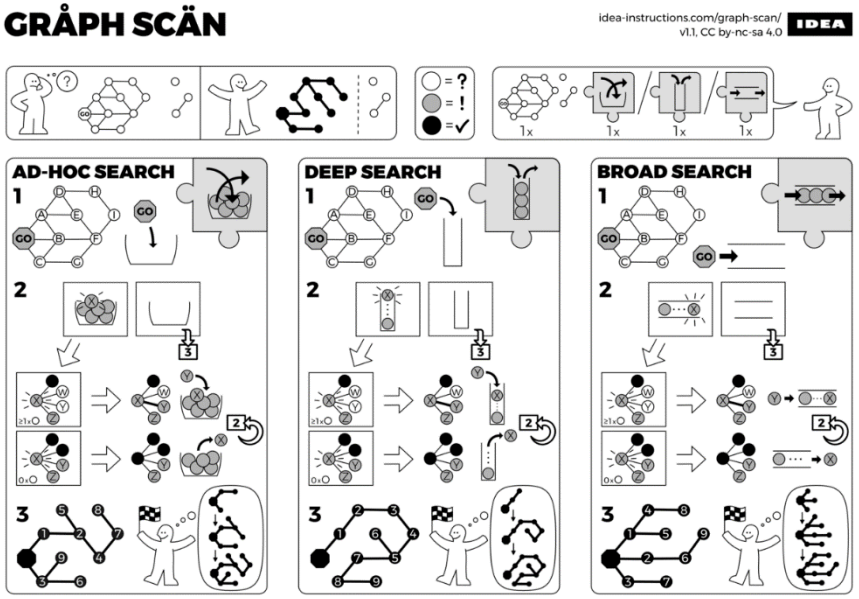


Content

- Introduction
- Basic Concepts
- Graph Abstract Data Type
- Graph Traversal
- Graph Applications
- Some Difficult Problems

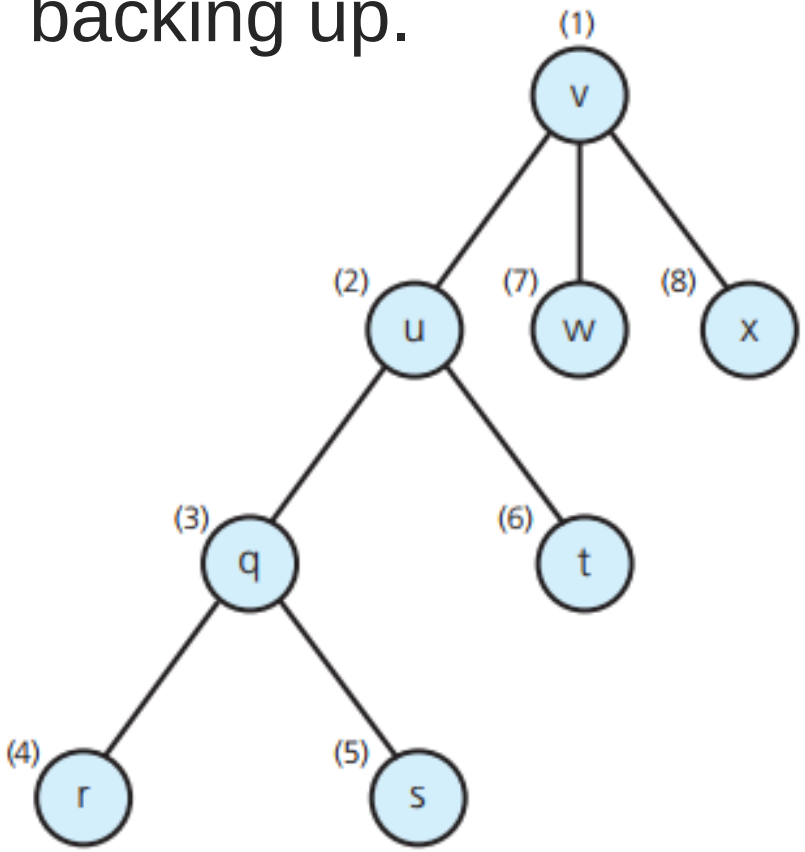
Graph Traversal

- A graph traversal that starts at vertex v will visit all vertices w for which there is a path between v and w.
  - If a graph is not connected, a graph traversal that begins at vertex v will visit only a subset of the graph's vertices. This subset is called the connected component containing v.
  - If a graph contains a cycle, a graph-traversal algorithm can loop indefinitely. To prevent such a misfortune, the algorithm must mark each vertex during a visit and must never visit a vertex more than once.



Depth-First Search

- From a given vertex v, the depth-first search (DFS) strategy of graph traversal proceeds along a path from v as deeply into the graph as possible before backing up.

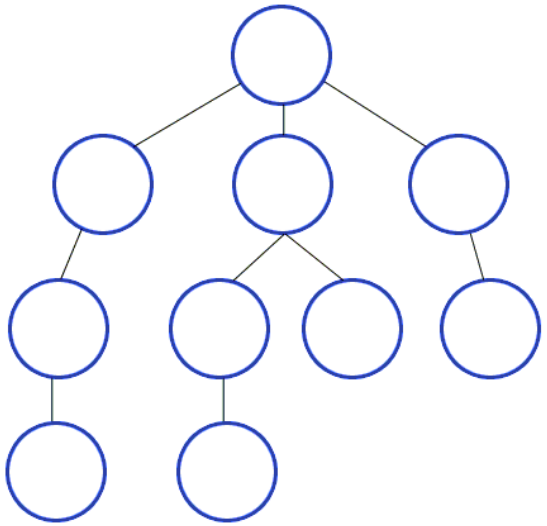


The DFS traversal algorithm marks and then visits each of the vertices v, u, q, and r. When the traversal reaches a vertex—such as r—that has no unvisited adjacent vertices, it backs up and visits, if possible, an unvisited adjacent vertex. Thus, the traversal backs up to q and then visits s.

DFS pseudocode

- The DFS strategy has a simple recursive form:

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Recursive version.
dfs(v: Vertex)
      Mark v as visited
      for (each unvisited vertex u adjacent to v)
            dfs(u)
```

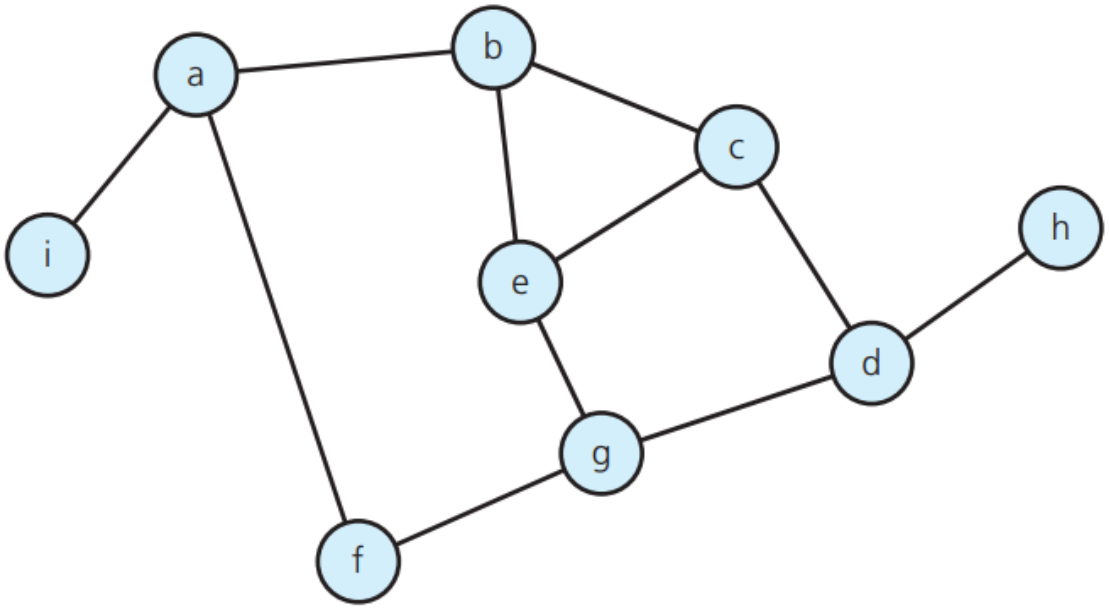


DFS pseudocode

- An iterative version of the DFS algorithm is also possible by using a stack:

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Iterative version.
dfs(v: Vertex)
        s= a new empty stack
        // Push v onto the stack and mark it
        s.push(v)
        Mark v as visited
        // Loop invariant: there is a path from vertex v at the
        // bottom of the stack s to the vertex at the top of s
        while (!s.isEmpty()) {
                if (no unvisited vertices are adjacent to the vertex on
                the top of the stack)
                        s.pop() // Backtrack
                else {
                        Select an unvisited vertex u adjacent to the
                        vertex on the top of the stack
                        s.push(u)
                        Mark u as visited
                }
        }
```

DFS pseudocode



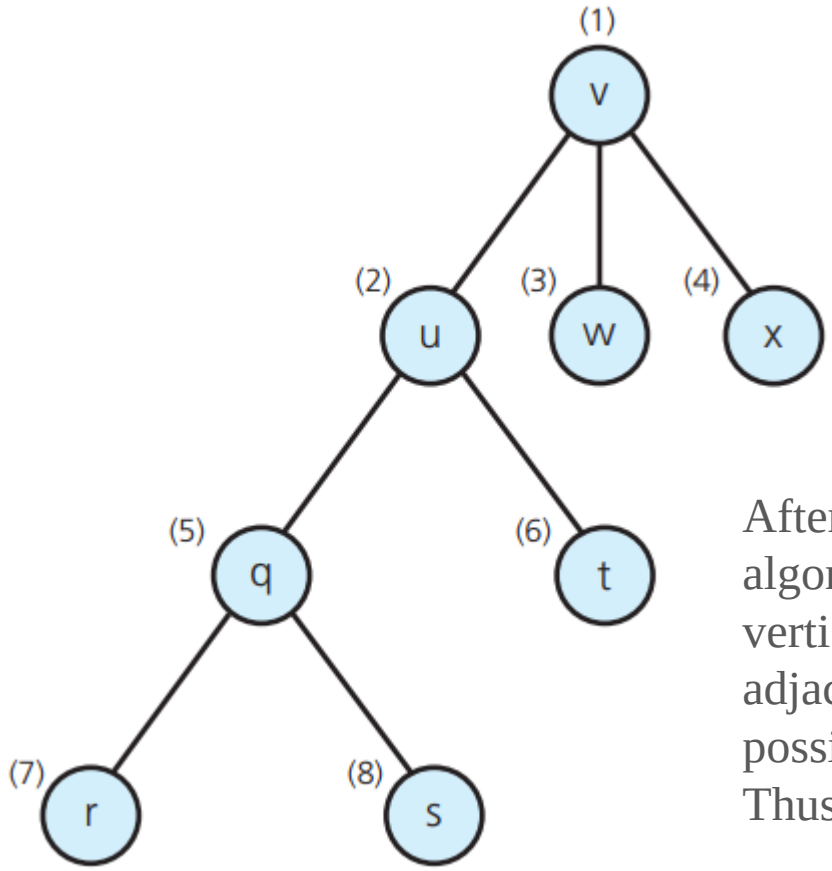
Beginning at vertex a, the traversal visits the vertices in this order: a, b, c, d, g, e, f, h, i.

Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

Demo in:
https://cs.stanford.edu/people/abisee/tutorial/dfs.html

Breadth-first search

- The breadth-first search (BFS) strategy of graph traversal visits every vertex adjacent to v that it can before visiting any other vertex.



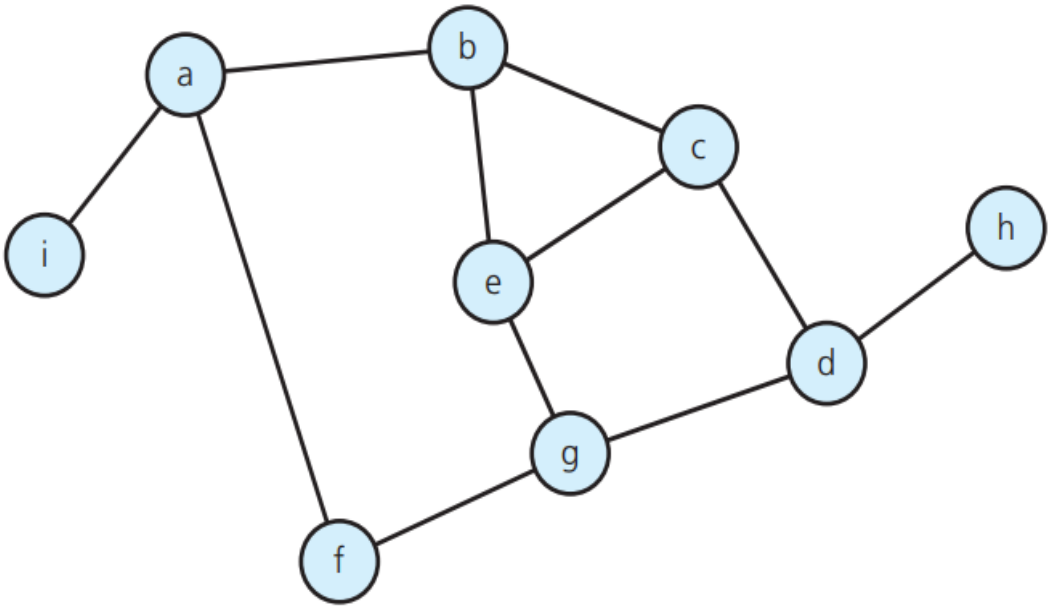
After marking and visiting v, the BFS traversal algorithm marks and then visits each of the vertices u, w, and x. Since no other vertices are adjacent to v, the BFS algorithm visits, if possible, all unvisited vertices adjacent to u. Thus, the traversal visits q and t.

BFS pseudocode

- An iterative version of this algorithm follows:

```
// Traverses a graph beginning at vertex v by using a
// breadth-first search: Iterative version.
bfs(v: Vertex)
        q = a new empty queue
        // Add v to queue and mark it
        q.enqueue(v)
        Mark v as visited
        while (!q.isEmpty()) {
                q.dequeue(w)
                // Loop invariant: there is a path from
                vertex w to every vertex in the queue q
                for (each unvisited vertex u adjacent to w)
                {
                        Mark u as visited
                        q.enqueue(u)
                }
        }
```

BFS pseudocode



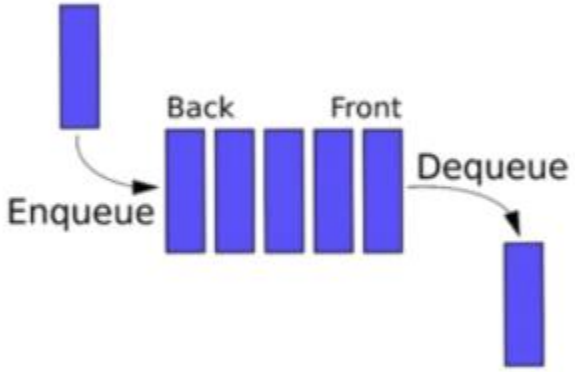
Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

The BFS traversal visits all of the vertices in this order: a, b, f, i, c, e, g, d, h

Demo in:
https://cs.stanford.edu/people/abisee/tutorial/bfs.html

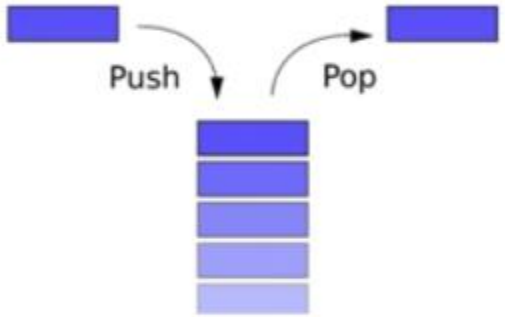
BFS vs DFS

BFS uses "first in first out".



This is a queue.

DFS uses "last in first out".

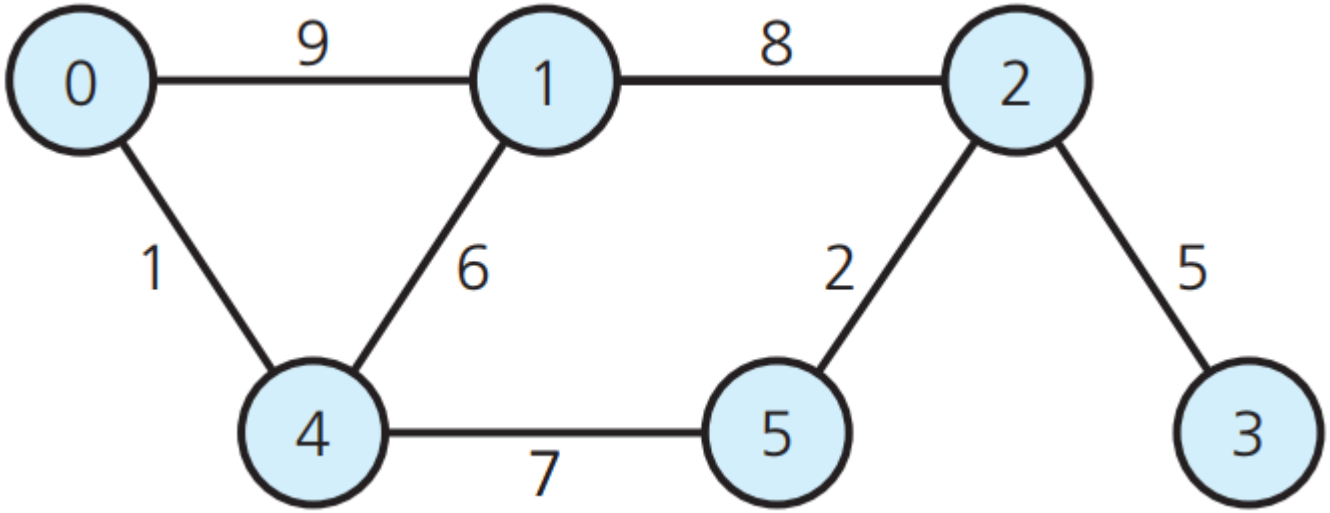


This is a stack.

Comparison demo in:
https://cs.stanford.edu/people/abisee/tutorial/bfsdfs.html

Exercises

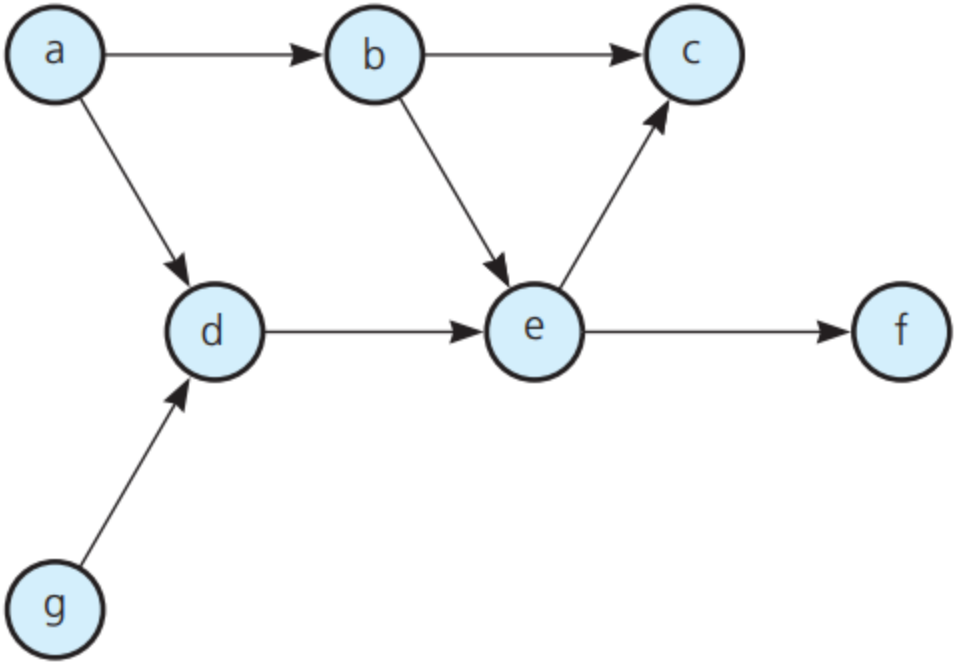
- Use the depth-first strategy and the breadth-first strategy to traverse the following graph, beginning with vertex 0. List the vertices in the order in which each traversal visits them



Content

- Introduction
- Basic Concepts
- Graph Abstract Data Type
- Graph Traversal
- Graph Applications
- Some Difficult Problems

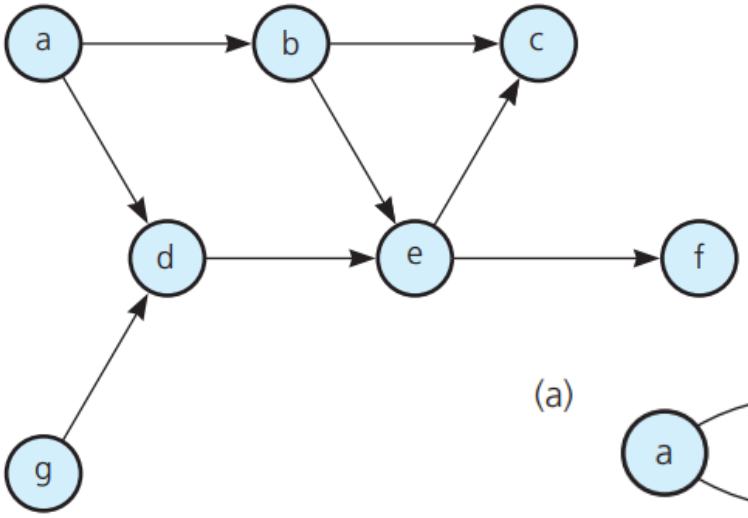
Applications of graphs 1



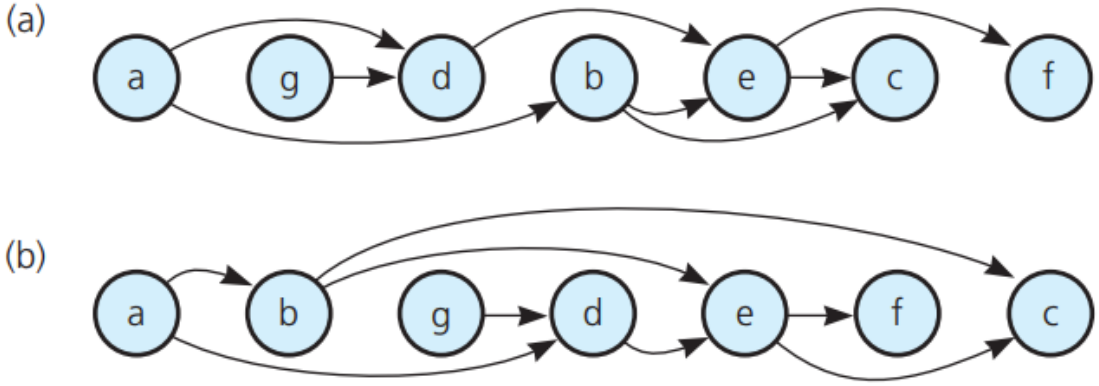
If the vertices represent academic courses, the graph represents the prerequisite structure for the courses. For example, course *a* is a prerequisite to course *b*, which is a prerequisite to both courses *c* and *e*. In what order should you take all seven courses so that you will satisfy all prerequisites?

Topological Sorting

- A directed graph without cycles has a natural order. It is a linear order, called a topological order.
- Arranging the vertices into a topological order is called topological sorting.



If you arrange the vertices of a directed graph linearly and in a topological order, the edges will all point in one direction.



Topological Sorting Algorithm 1

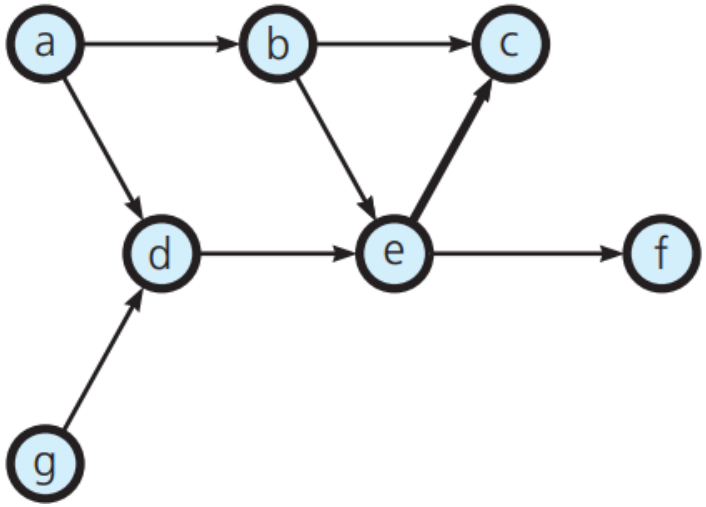
- Step 1: find a vertex that has no successor.
- Step 2: remove from the graph this vertex and all edges that lead to it, and add it to the beginning of a list of vertices.
- Step 3: repeat step 2 until the graph is empty. The list of vertices will be in topological order.

```
// Arranges the vertices in graph theGraph into a
// topological order and places them in list aList.
topSort1(theGraph: Graph, aList: List)
      n = number of vertices in theGraph
      for (step = 1 through n) {
            Select a vertex v that has no successors
            aList.insert(1, v)
            Remove from theGraph vertex v and its edges
      }
```

Topological Sorting Algorithm 1

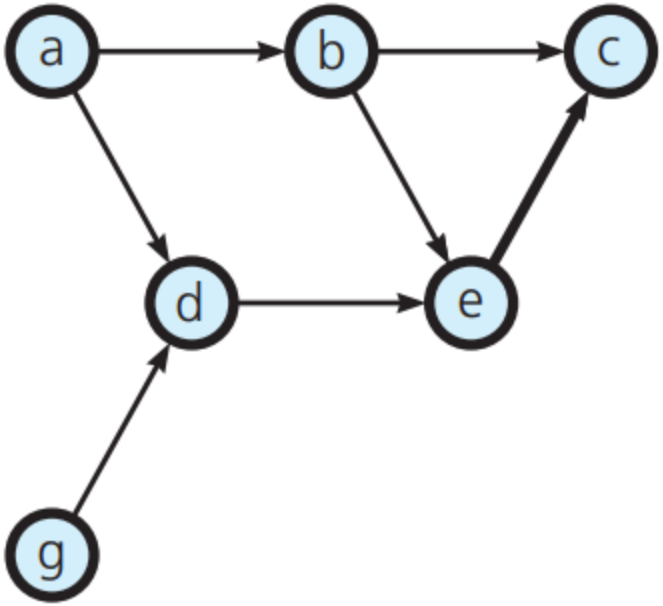
Graph `theGraph`

List `aList`



Topological Sorting Algorithm 1

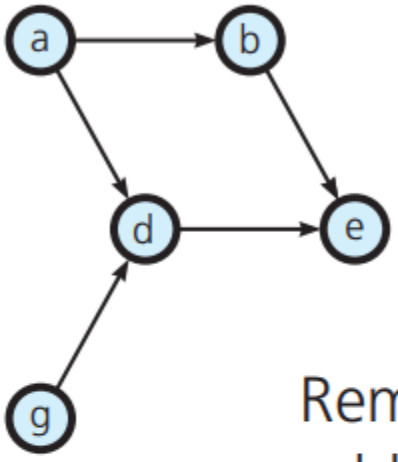
Remove f from `theGraph`;
add it to `aList`



f

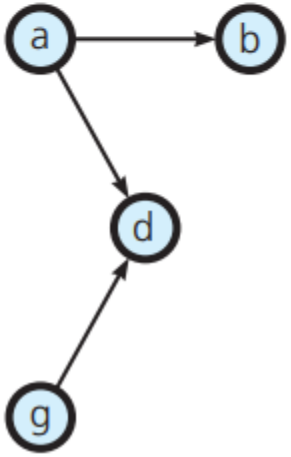
Topological Sorting Algorithm 1

Remove c from `theGraph`;
add it to `aList`



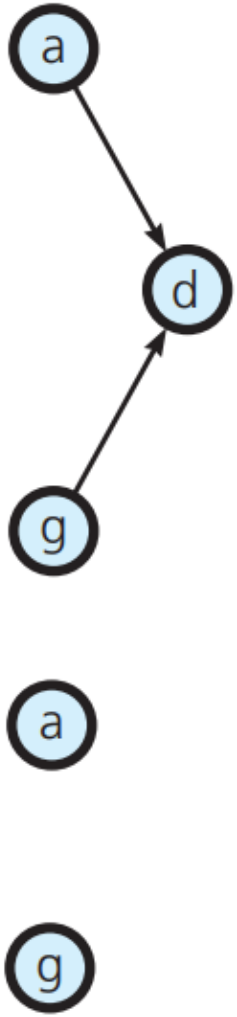
c f

Remove e from `theGraph`;
add it to `aList`



e c f

Topological Sorting Algorithm 1



Remove b from `theGraph`;
add it to `aList`

b e c f

Remove d from `theGraph`;
add it to `aList`

d b e c f

Topological Sorting Algorithm 1

Remove g from `theGraph`;
add it to `aList`

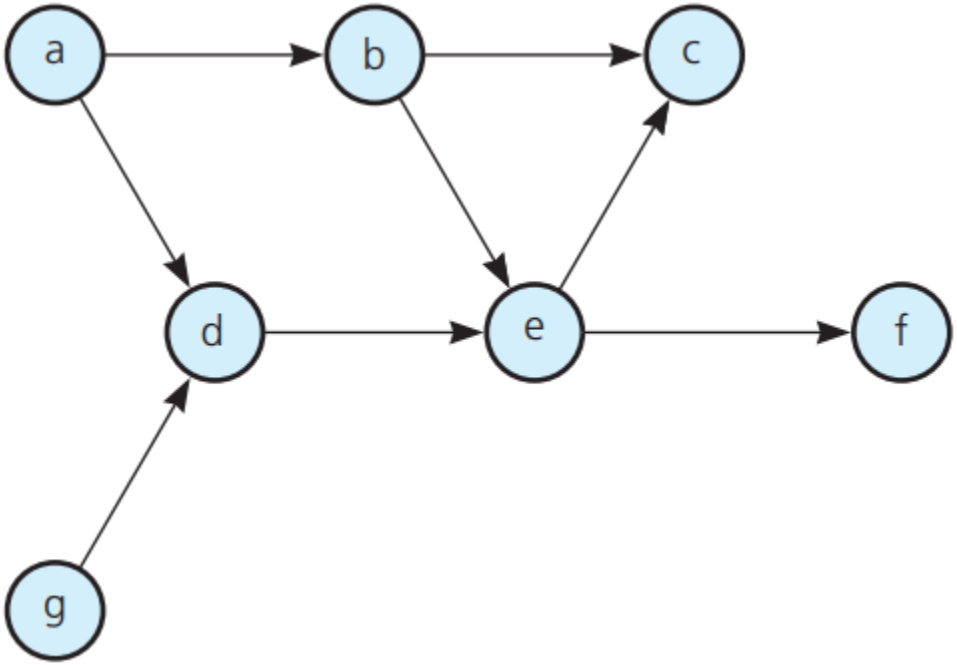
g d b e c f

Remove a from `theGraph`;
add it to `aList`

a g d b e c f

Topological Sorting Algorithm 2

- Another algorithm is a simple modification of the iterative depth-first search algorithm.
  - First you push all vertices that have no predecessor onto a stack.
  - Each time you pop a vertex from the stack, you add it to the beginning of a list of vertices.



Topological Sorting Algorithm

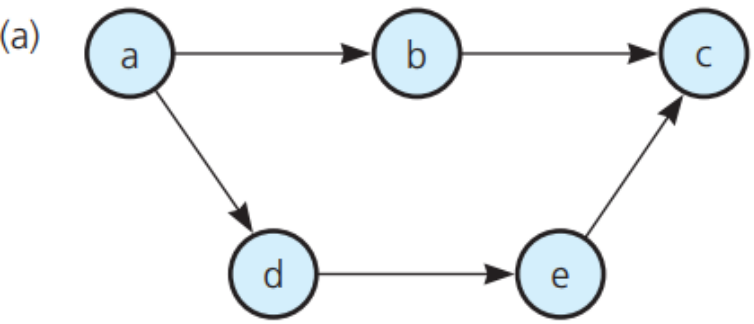
```
topSort2(theGraph: Graph, aList: List)
        s = a new empty stack
        for (all vertices v in the graph)
                if (v has no predecessors) {
                        s.push(v)
                        Mark v as visited
                }

        while (!s.isEmpty()) {
                if (all vertices adjacent to the vertex on the
                        top of the stack have been visited) {
                        s.pop(v)
                        aList.insert(1, v)
                } else {
                        Select an unvisited vertex u adjacent to
                        the vertex on the top of the stack
                        s.push(u)
                        Mark u as visited
                }
        }
```

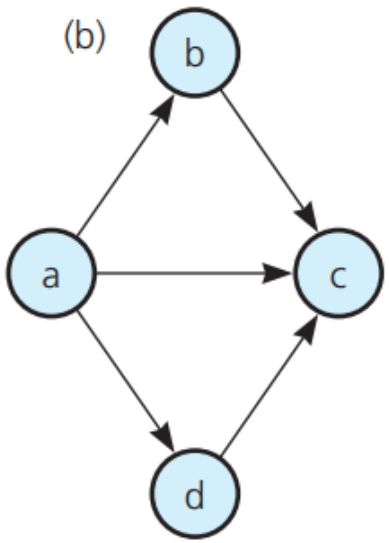
Topological Sorting Algorithm

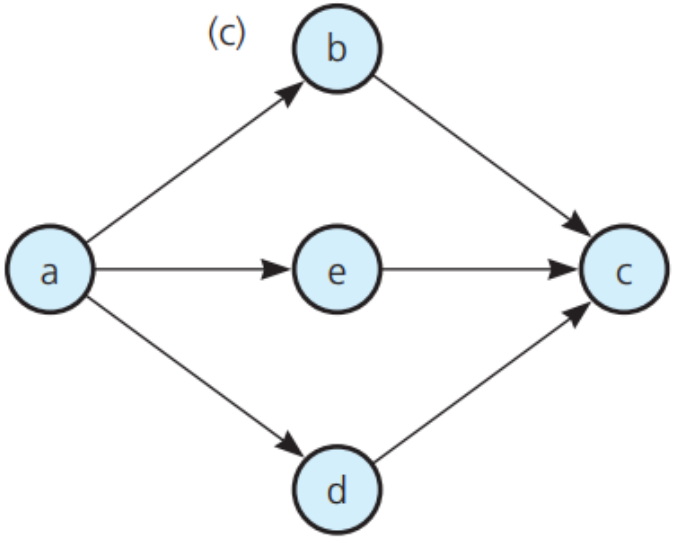
Action	Stack `s` (bottom to top)	List `aList` (beginning to end)
Push a	a	
Push g	a g	
Push d	a g d	
Push e	a g d e	
Push c	a g d e c	
Pop c, add c to `aList`	a g d e	c
Push f	a g d e f	c
Pop f, add f to `aList`	a g d e	f c
Pop e, add e to `aList`	a g d	e f c
Pop d, add d to `aList`	a g	d e f c
Pop g, add g to `aList`	a	g d e f c
Push b	a b	g d e f c
Pop b, add b to `aList`	a	b g d e f c
Pop a, add a to `aList`	(empty)	a b g d e f c

Exercise

- Write all possible topological orders for the vertices in the following graph.

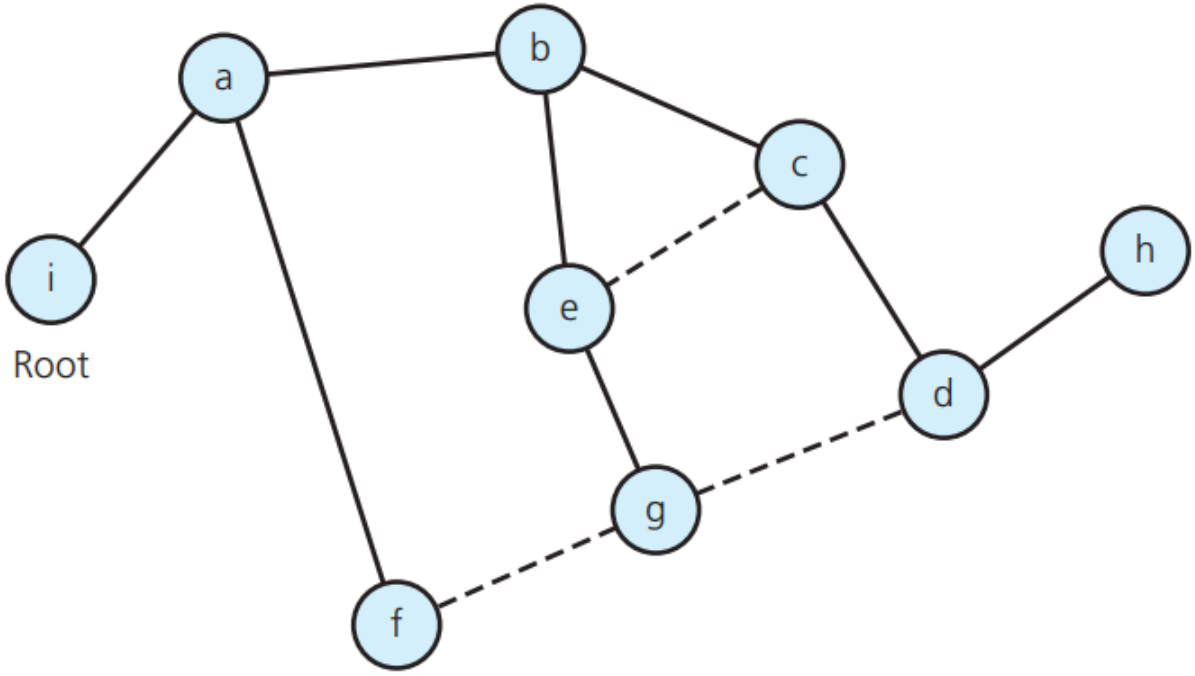






Applications of graphs 2

- A spanning tree of a connected undirected graph G is a subgraph of G that contains all of G's vertices and enough of its edges to form a tree.

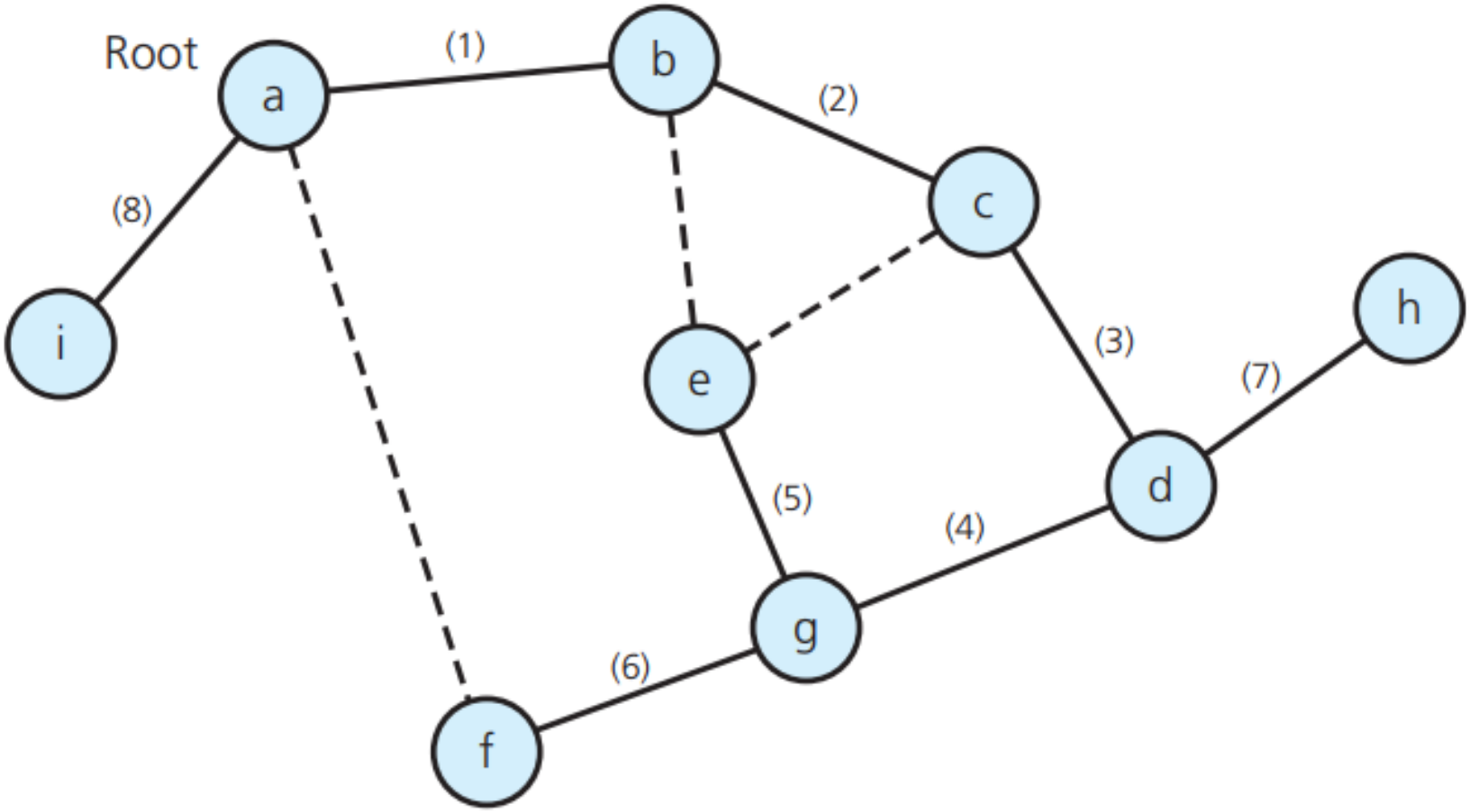


DFS Spanning Tree Algorithm

- With a depth-first search, as you traverse the graph, mark the edges that you follow. After the traversal is complete, the graph's vertices and marked edges form a spanning

```
// Forms a spanning tree for a connected undirected graph
// beginning at vertex v by using depth-first search:
// Recursive version.
dfsTree(v: Vertex)
      Mark v as visited
      for (each unvisited vertex u adjacent to v) {
            Mark the edge from u to v
            dfsTree(u)
      }
```

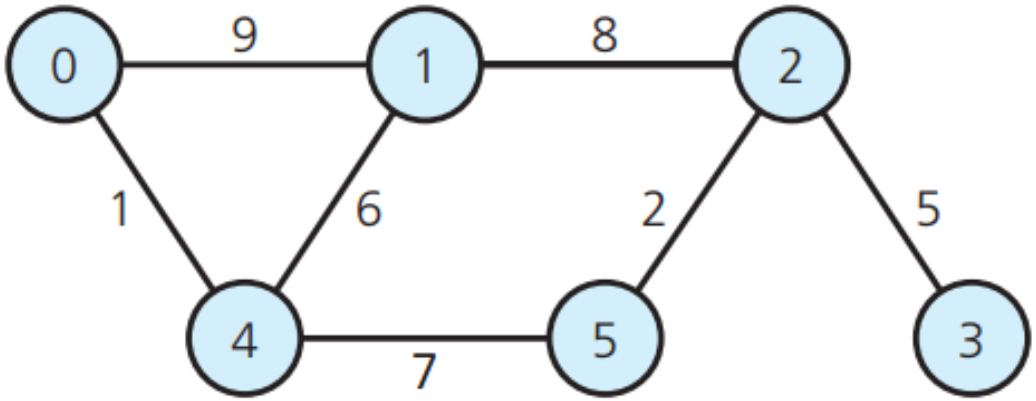
DFS Spanning Tree Algorithm



The DFS spanning tree algorithm visits vertices in this order: a, b, c, d, g, e, f, h, i. Numbers indicate the order in which the algorithm marks edges.

Exercies

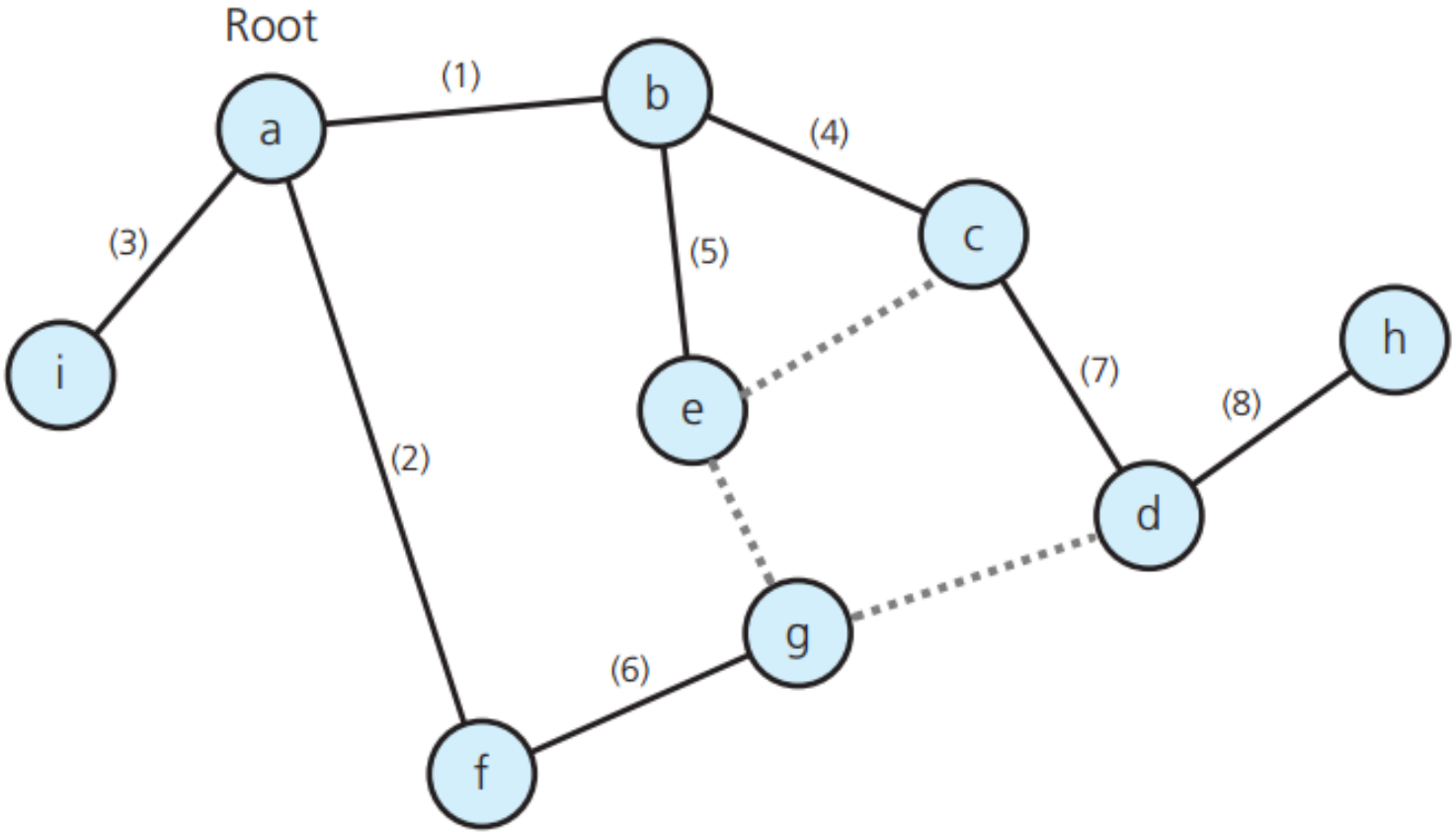
- Draw the DFS spanning tree whose root is vertex 0 for the graph:



BFS Spanning Tree Algorithm

```
// Forms a spanning tree for a connected undirected graph
// beginning at vertex v by using breadth-first search:
// Iterative version.
bfsTree(v: Vertex)
      q = a new empty queue
      // Add v to queue and mark it
      q.enqueue(v)
      Mark v as visited
      while (!q.isEmpty()) {
            q.dequeue(w)
            //Loop invariant:there is a path from vertex w
            //to every vertex in the queue q
            for (each unvisited vertex u adjacent to w) {
                  Mark u as visited
                  Mark edge between w and u
                  q.enqueue(u)
            }
      }
```

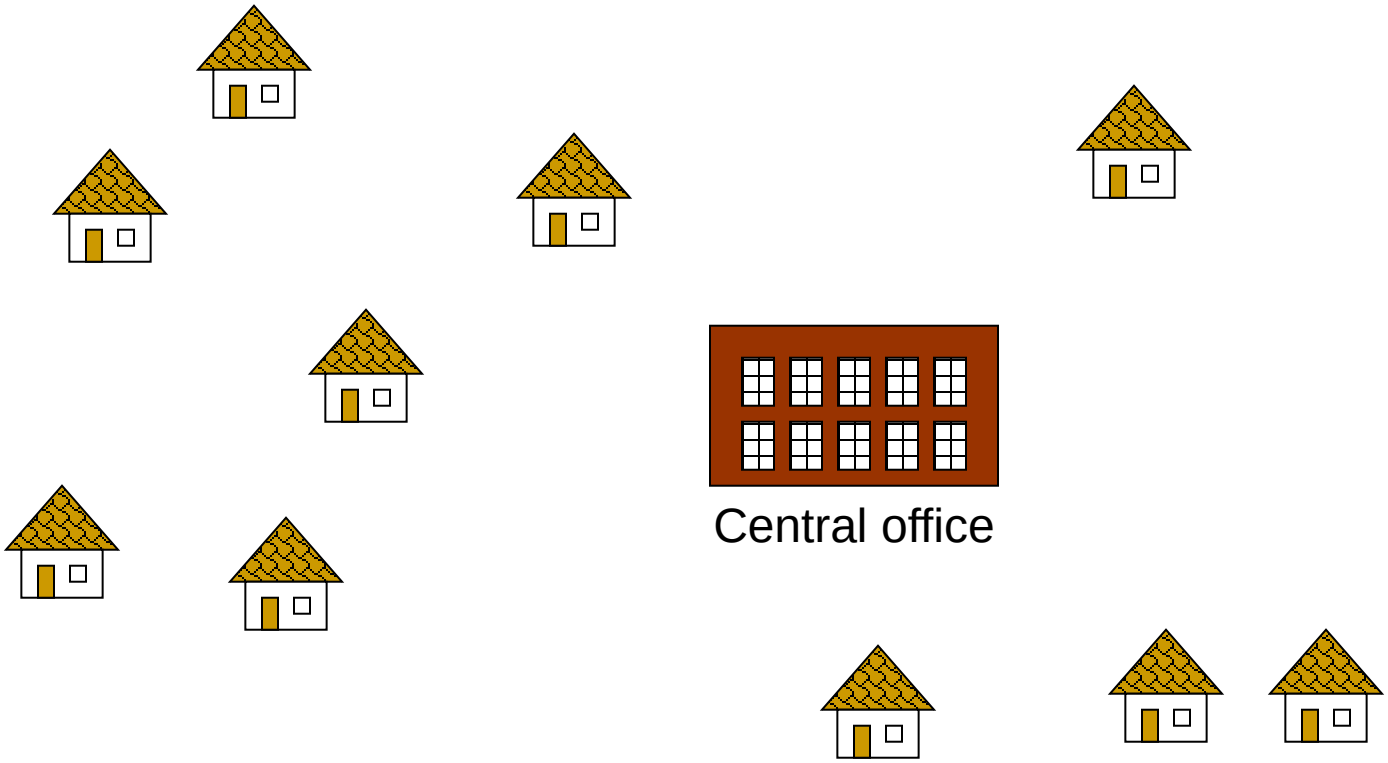
BFS Spanning Tree Algorithm



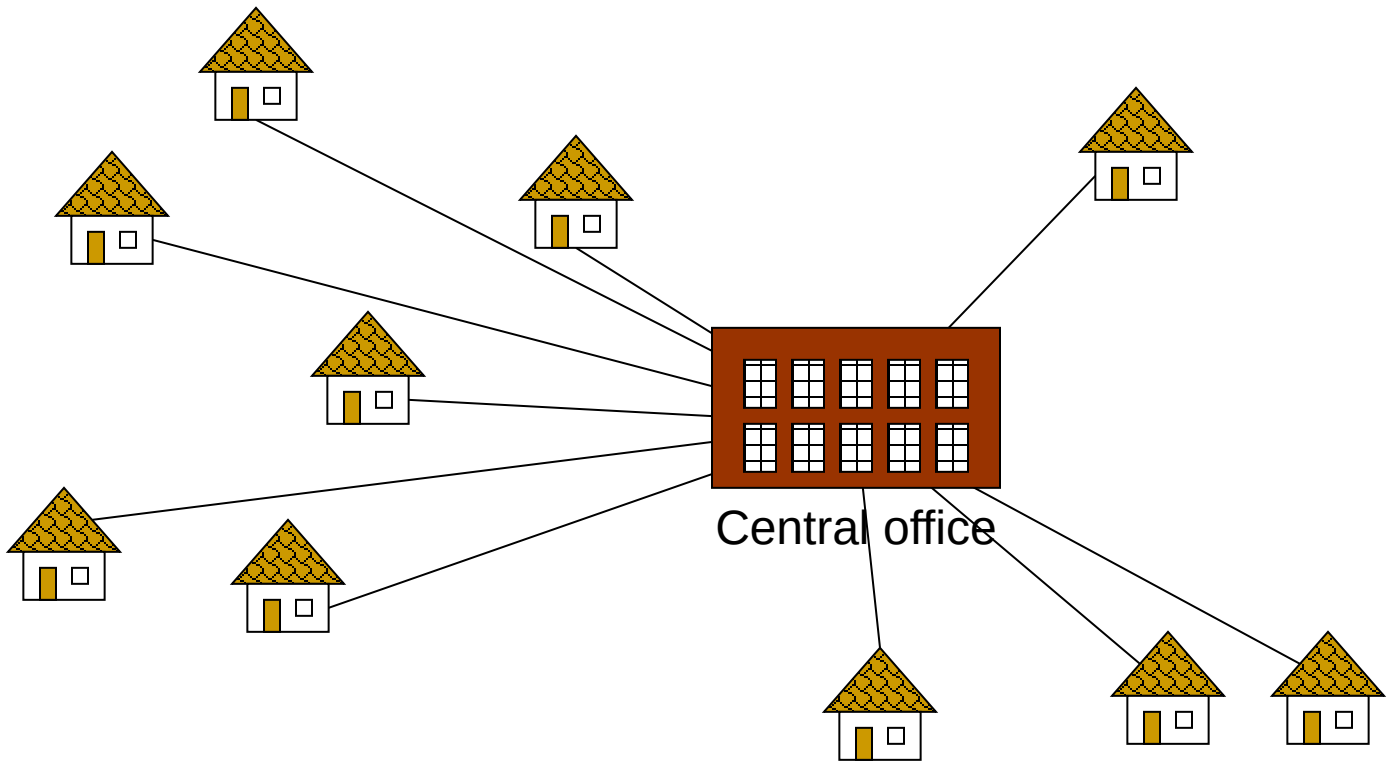
The BFS spanning tree algorithm visits vertices in this order: a, b, f, i, c, e, g, d, h. Numbers indicate the order in which the algorithm marks edges.

Applications of graphs 3

- Imagine that a developing country hires you to design its telephone system so that all the cities in the country can call one another.

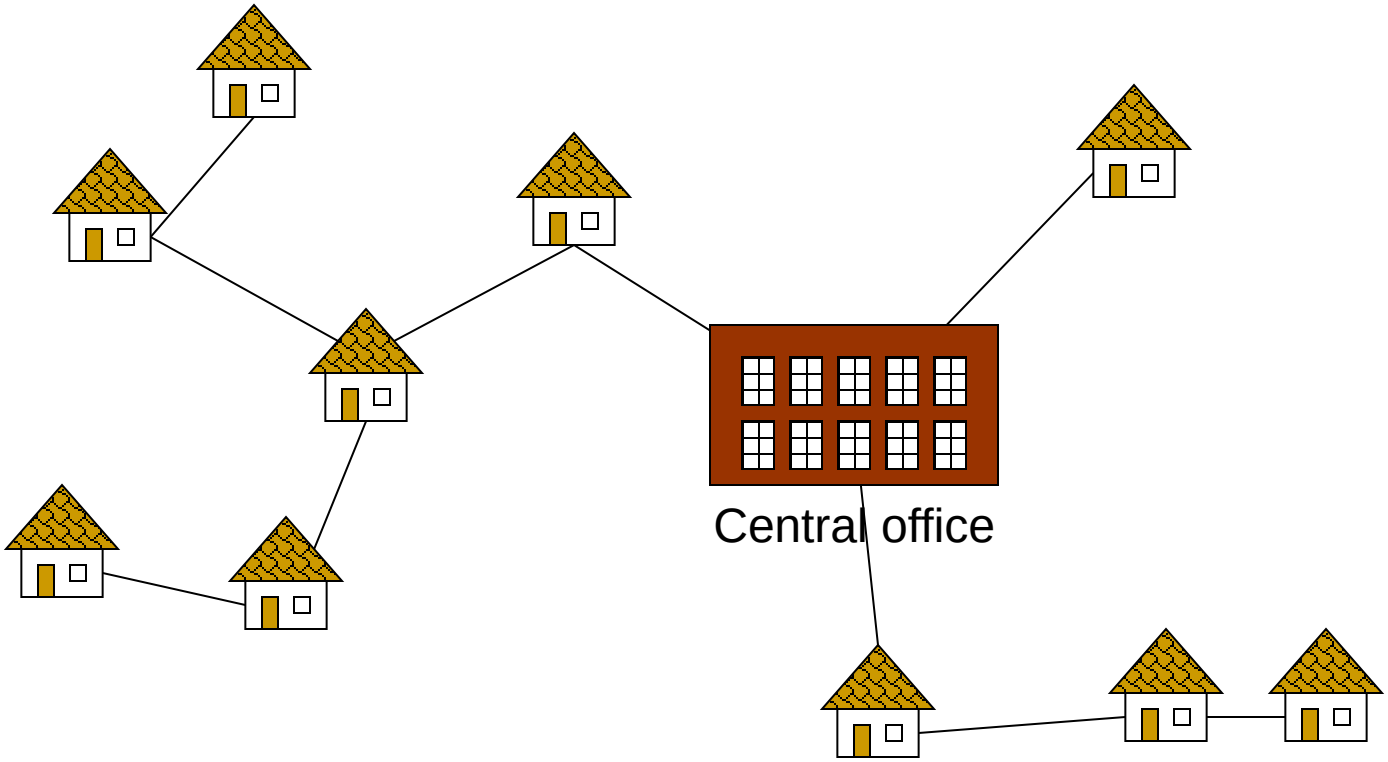


Applications of graphs 3 (cont.)



Expensive!

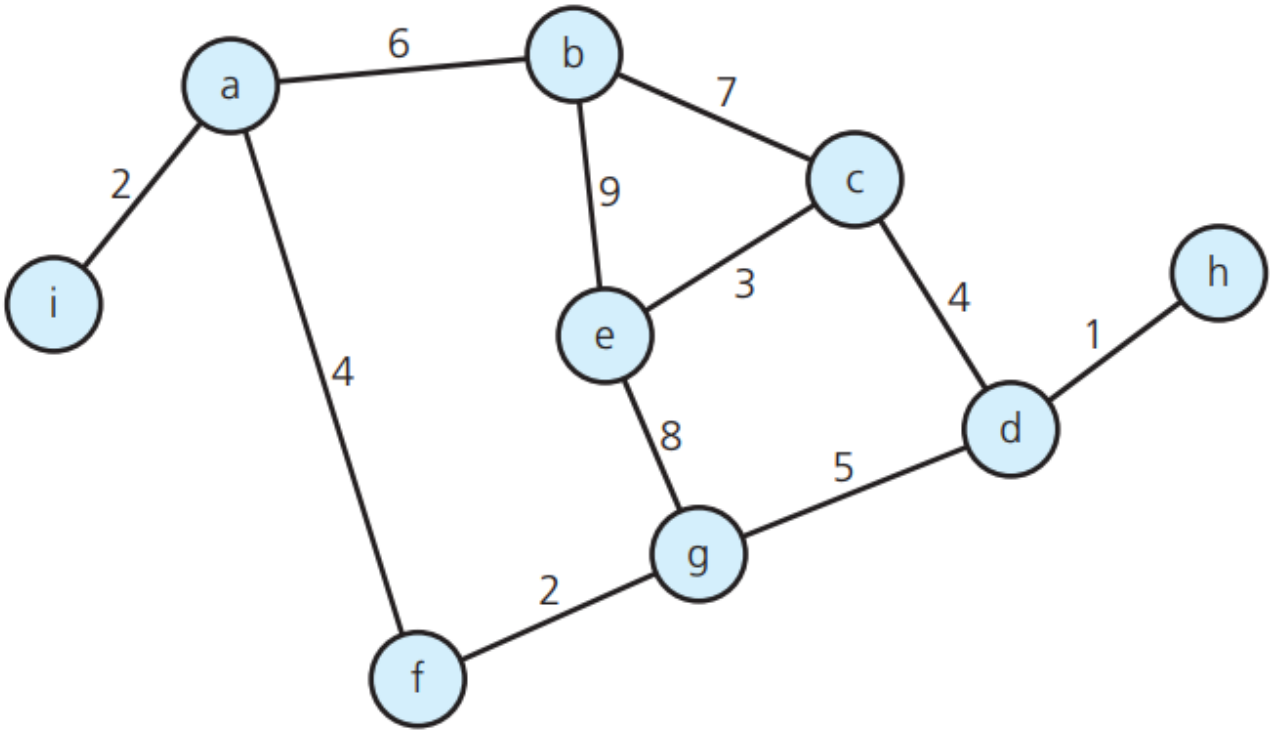
Applications of graphs 3 (cont.)



Minimize the total length of wire connecting **ALL** customers

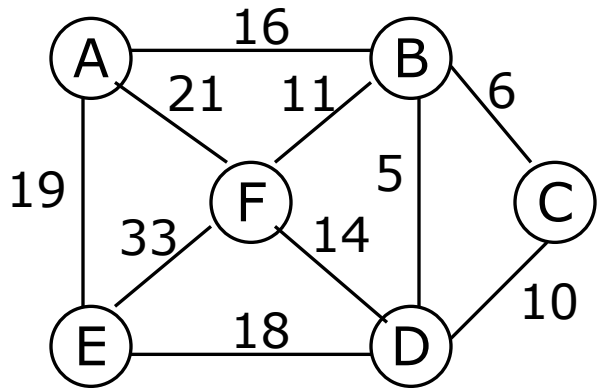
Applications of graphs 3 (cont.)

- In the following figure:
  - The vertices in the graph represent *n* cities.
  - An edge between two vertices indicates that it is place a telephone line between the cities that the vertices represent
  - Each edge's weight represents the installation cost of the telephone line.

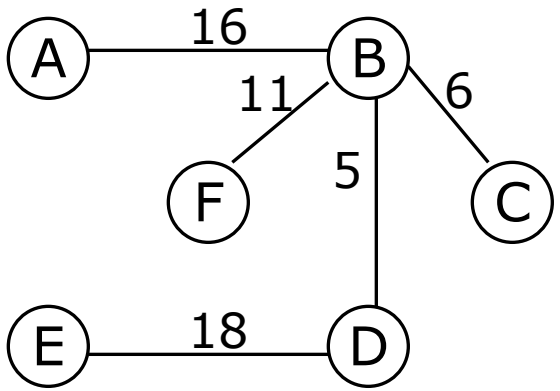


Minimum Spanning Tree

- We know that:
  - There may be more than one spanning tree
  - The cost of different trees may vary
- Solve the problem by selecting a spanning tree with the least cost (sum of the edge weights (costs) is minimal) → minimum spanning tree (MST).
- Although there may be several minimum spanning trees for a particular graph, their costs are equal.



A connected, undirected graph



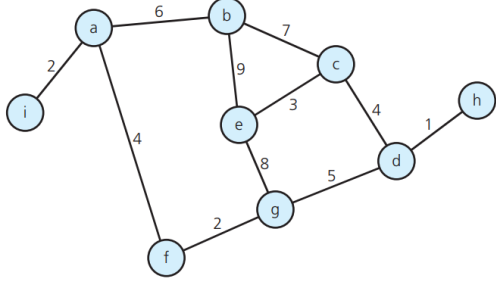
A minimum-cost spanning tree

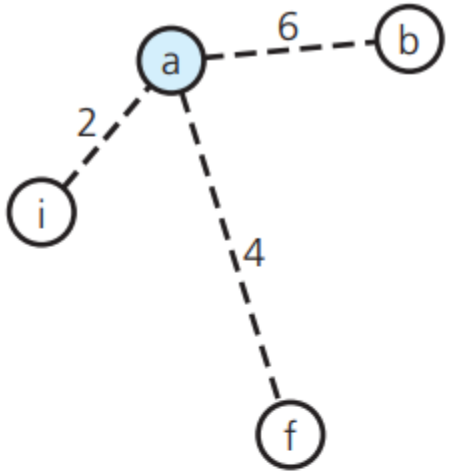
Prim's algorithm

- Step 1: initialize the tree with any vertex.
- Step 2: select a least-cost edge from among edges which begin with a vertex in the current tree and end with a vertex not in the current tree.
- Step 3: add this least-cost edge to tree and mark a new vertex as visited.
- Step 4: repeat step 2 until there are not unvisisted vertices.

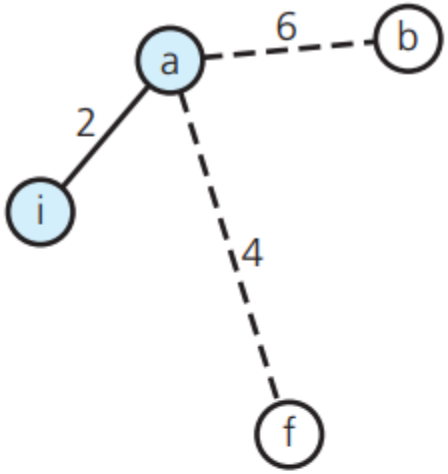
```
// Determines a minimum spanning tree for a weighted,
// connected, undirected graph whose weights are
// nonnegative, beginning with any vertex v.
primsAlgorithm(v: Vertex)
        Mark vertex v as visited and include it in the MST
        while (there are unvisited vertices) {
                Find the least-cost edge (v, u) from a visited
                vertex v to some unvisited vertex u
                Mark u as visited
                Add the vertex u and the edge (v, u) to the MST
        }
```

Prim's algorithm

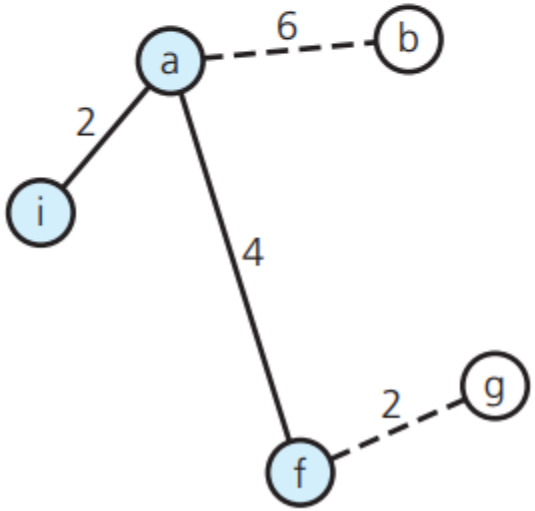




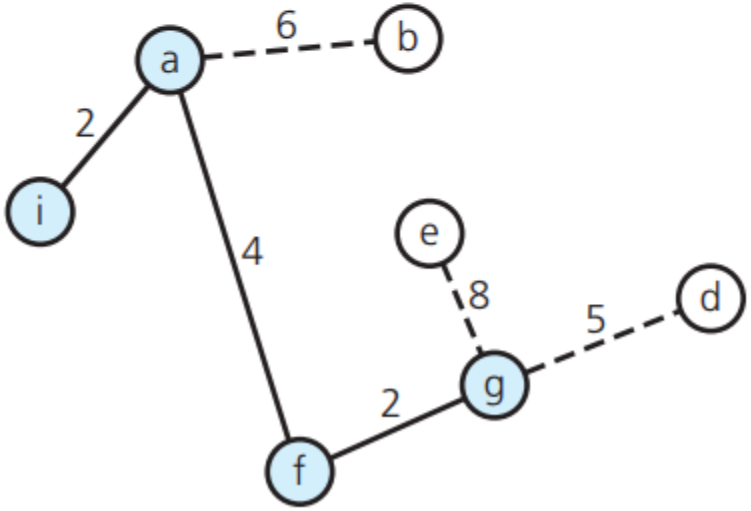
(a) Mark a, consider edges from a



(b) Mark i, include edge (a, i)

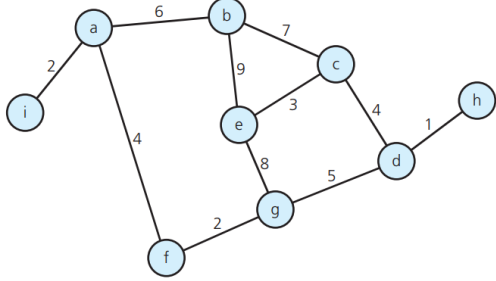


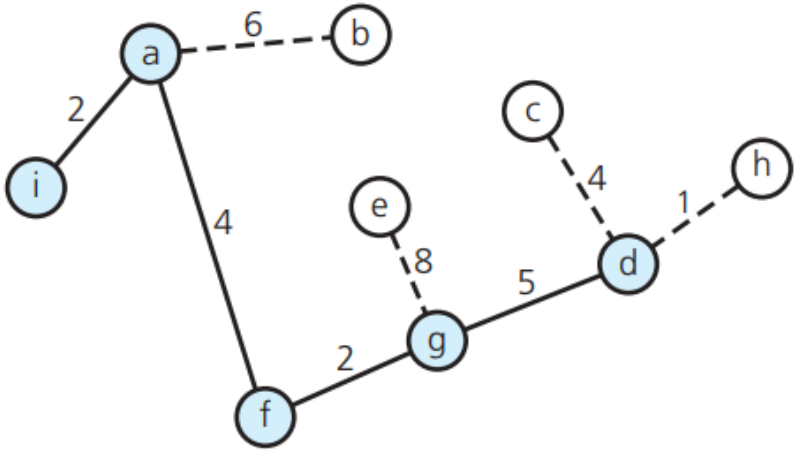
(c) Mark f, include edge (a, f)



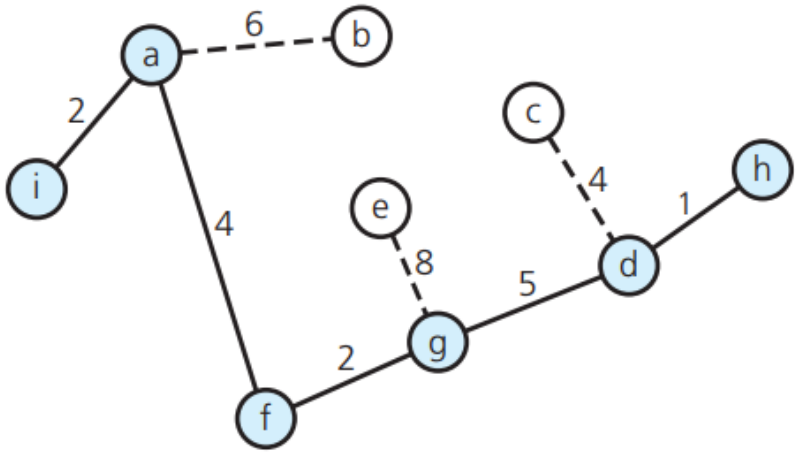
(d) Mark g, include edge (f, g)

Prim's algorithm

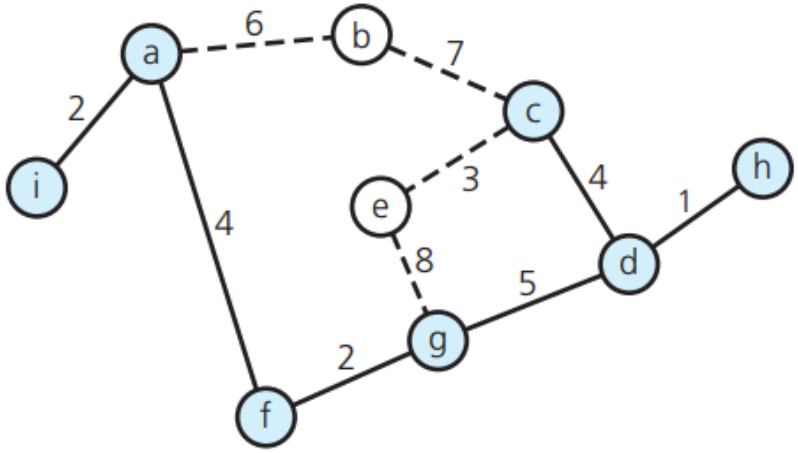




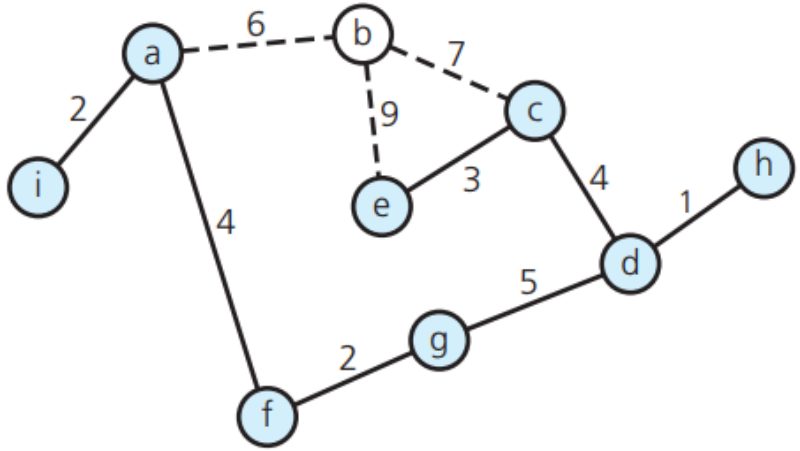
(e) Mark d, include edge (g, d)



(f) Mark h, include edge (d, h)

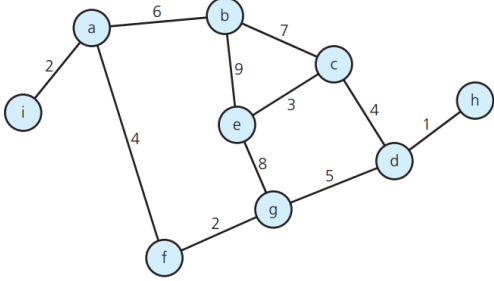


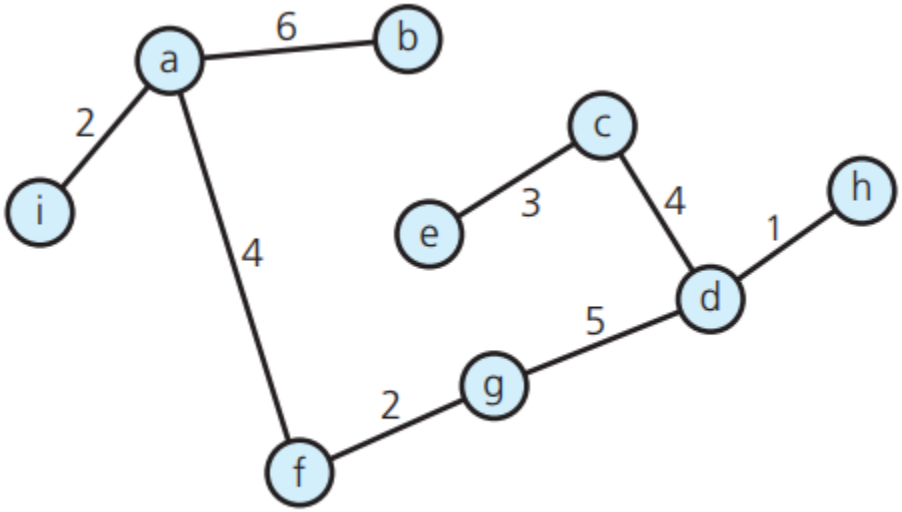
(g) Mark c, include edge (d, c)



(h) Mark e, include edge (c, e)

Prim's algorithm

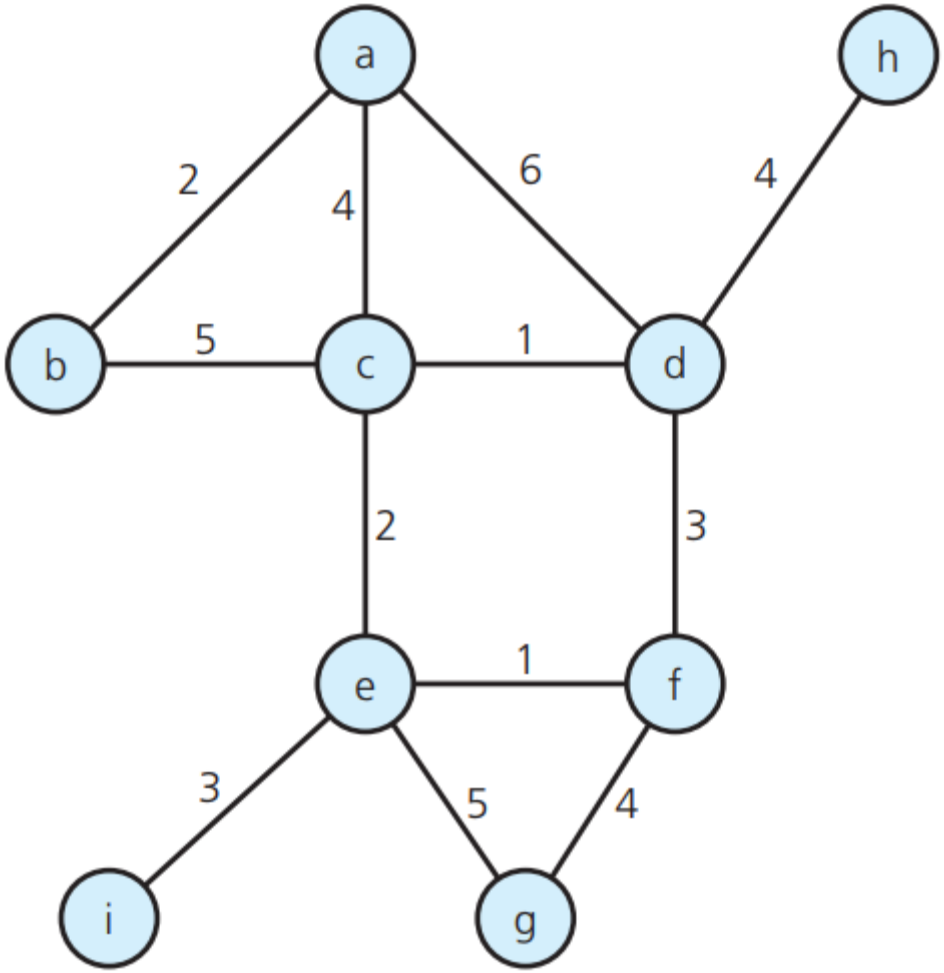




(i) Mark b, include edge (a, b)

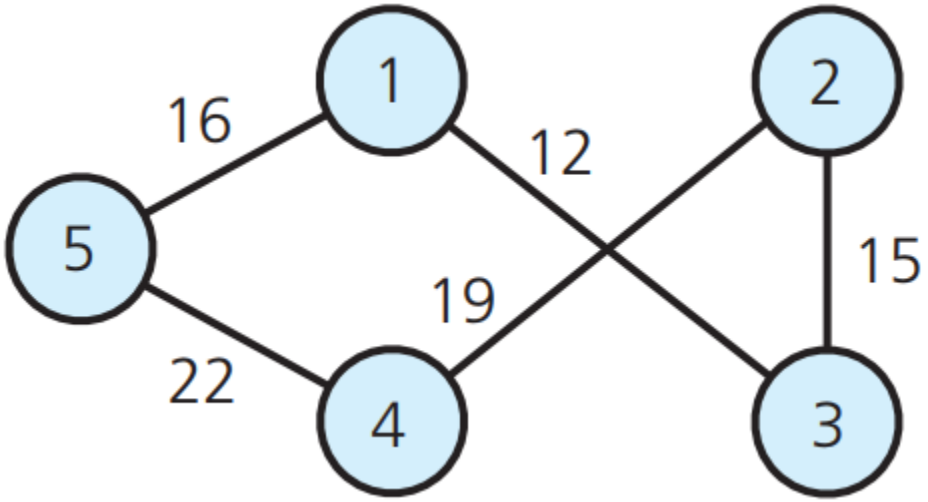
Exercises

- Draw the DFS and BFS spanning trees rooted at *a* for the following graph.
- Then draw the minimum spanning tree rooted at *a* for this graph.



Exercises

- For the graph in following figure:
  - Draw all the possible spanning trees.
  - Draw the minimum spanning tree.



Applications of graphs 4

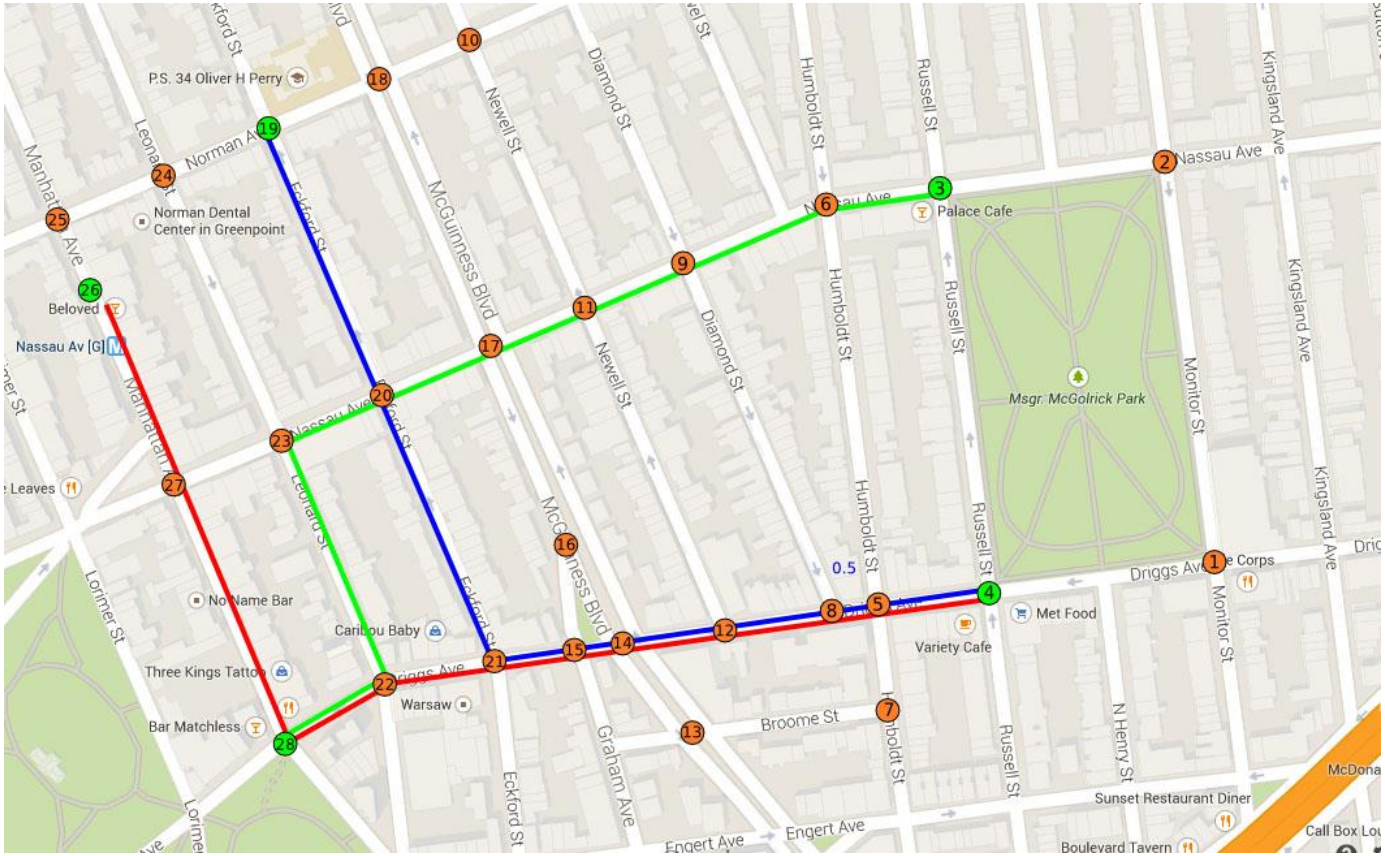
- We’re required to find the shortest path between two particular vertices in city map.

Small section of Williamsburg in Brooklyn, NY

http://www.marcinkossakowski.com/finding-shortest-path-using-dijkstras-algorithm/

Shortest Path

- The shortest path between two given vertices in a weighted graph is the path that has the smallest sum of its edge weights.

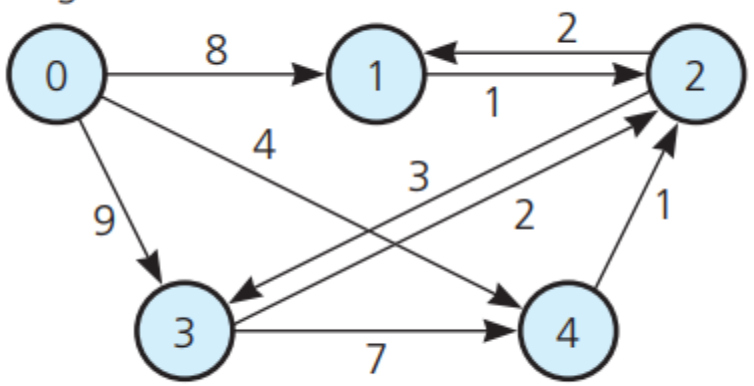


Test 1/Red: 4 -> 26
Test 2/Blue: 4 -> 19
Test 3/Green: 3 -> 28

Result of Dijkstra's algorithm directions

Shortest Path

(a) Origin



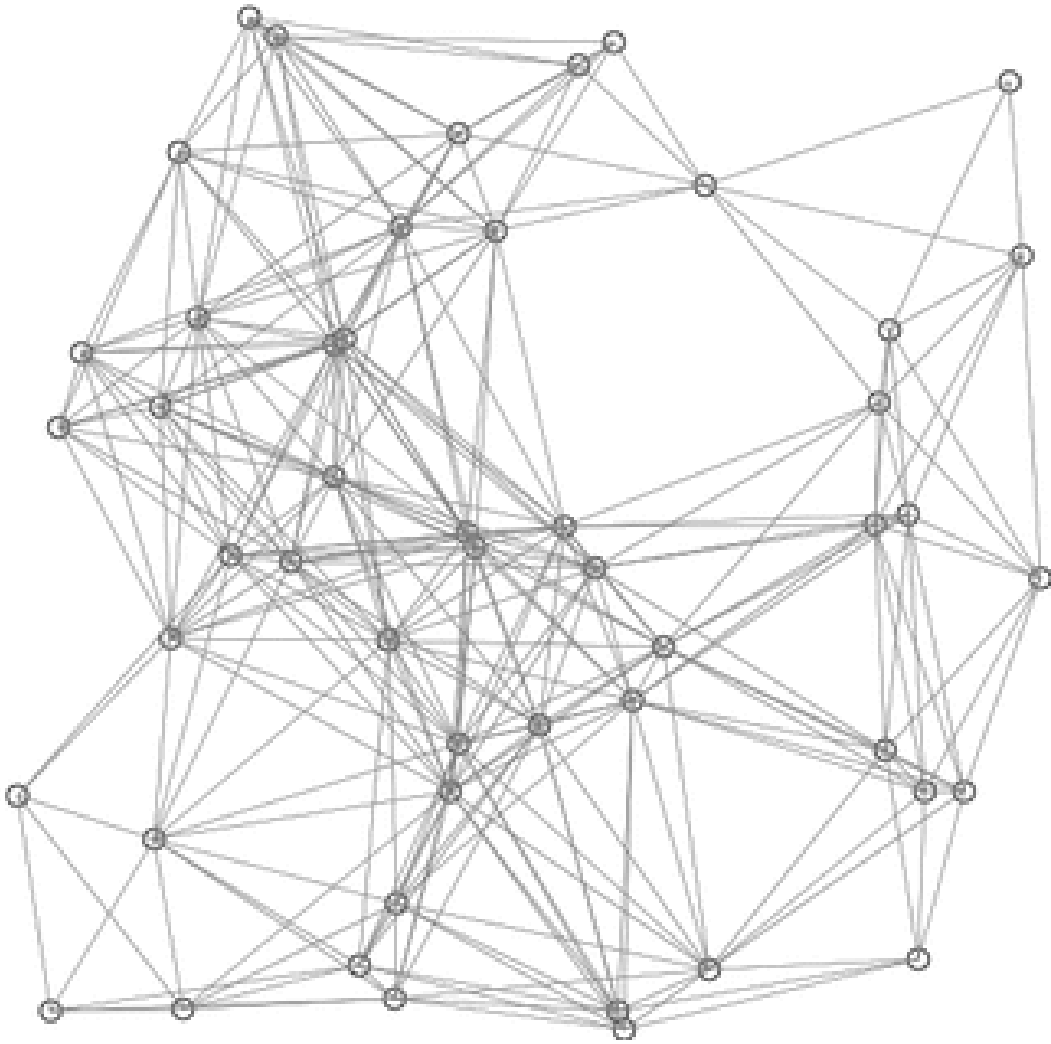
(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

The shortest path from vertex 0 to vertex 1 in the graph in above figure is not the edge between 0 and 1—its cost is 8—but rather the path from 0 to 4 to 2 to 1, with a cost of 7.

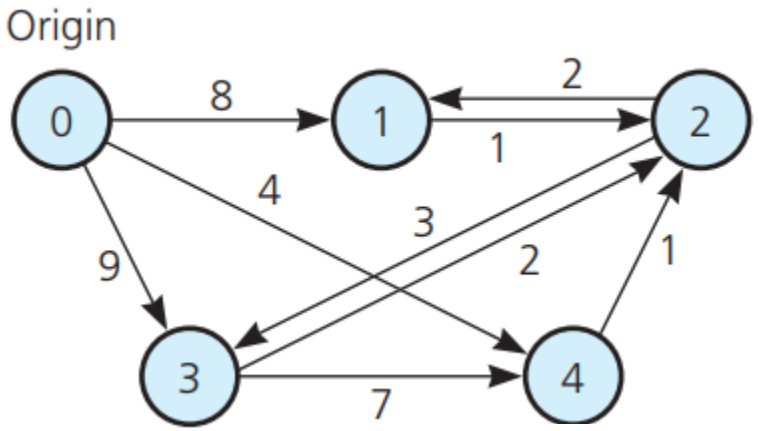
E. Dijkstra's Algorithm

- The Dijkstra's algorithm determines the shortest paths between a given origin and all other vertices.
- The algorithm uses a set *vertexSet* of selected vertices and an array weight, where *weight[v]* is the weight of the shortest path from vertex 0 to vertex v.



E. Dijkstra's Algorithm

- Step 1: vertexSet initially contains vertex 0, and weight is initially the first row of the graph's adjacency matrix.
  - weight contains the weights of the single-edge paths from vertex 0 to all other vertices.
  - That is, *weight[v]* equals *matrix[0][v]* for all v



			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4

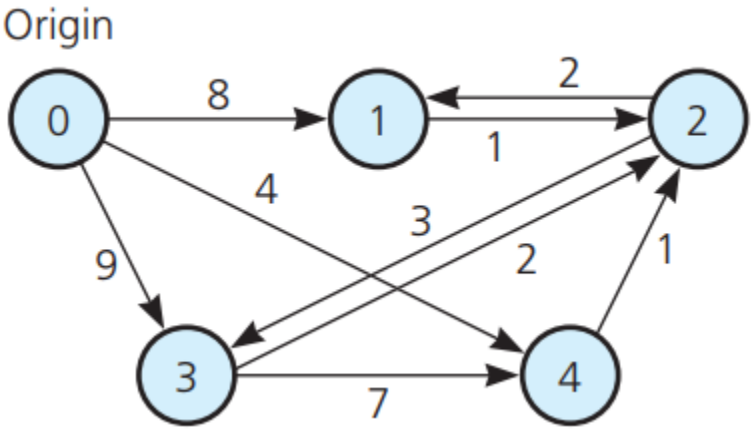
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

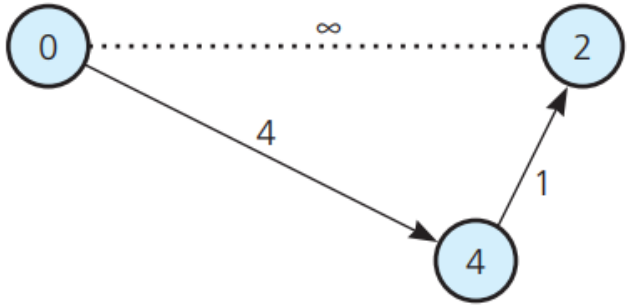
E. Dijkstra's Algorithm

- Step 2: Find a vertex v that is not in *vertexSet* and has weight as smallest. After that:
  - Add v to *vertexSet*
  - For all u which have edge from v to u, recalculate weight[u] with:

  weight[u] = **min** (*current weight[u]*, *weight[v] + matrix[v][u]*)

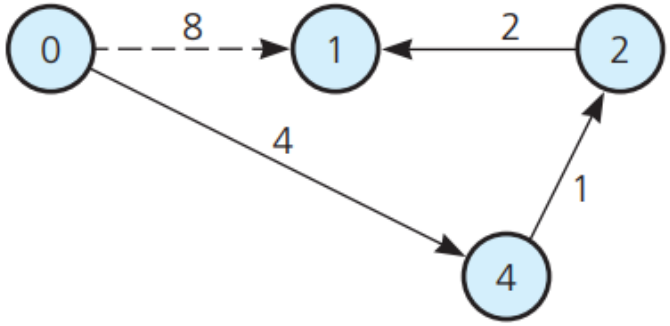
Step	v	vertexSet	weight [0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4

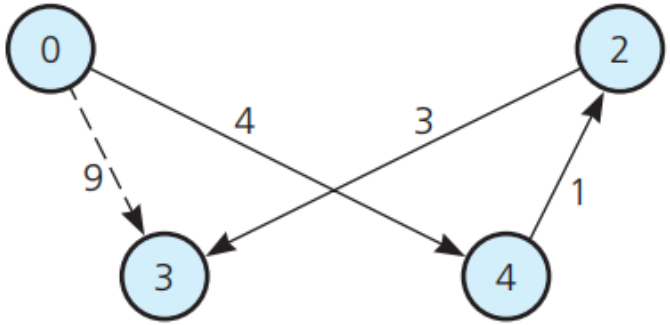


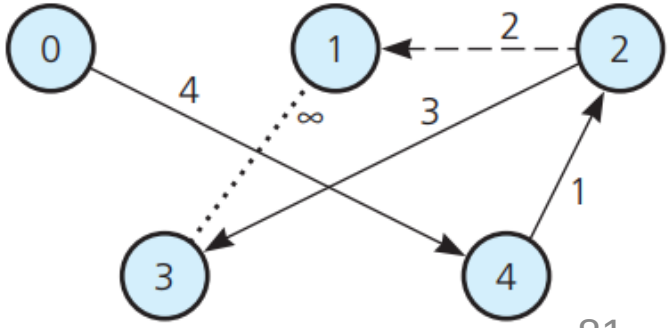


E. Dijkstra's Algorithm

- Step 3: repeat step 2 until a destination vertex x is placed in *vertexSet*.
  - weight[x] is the weight of the absolutely shortest path from 0 to x.



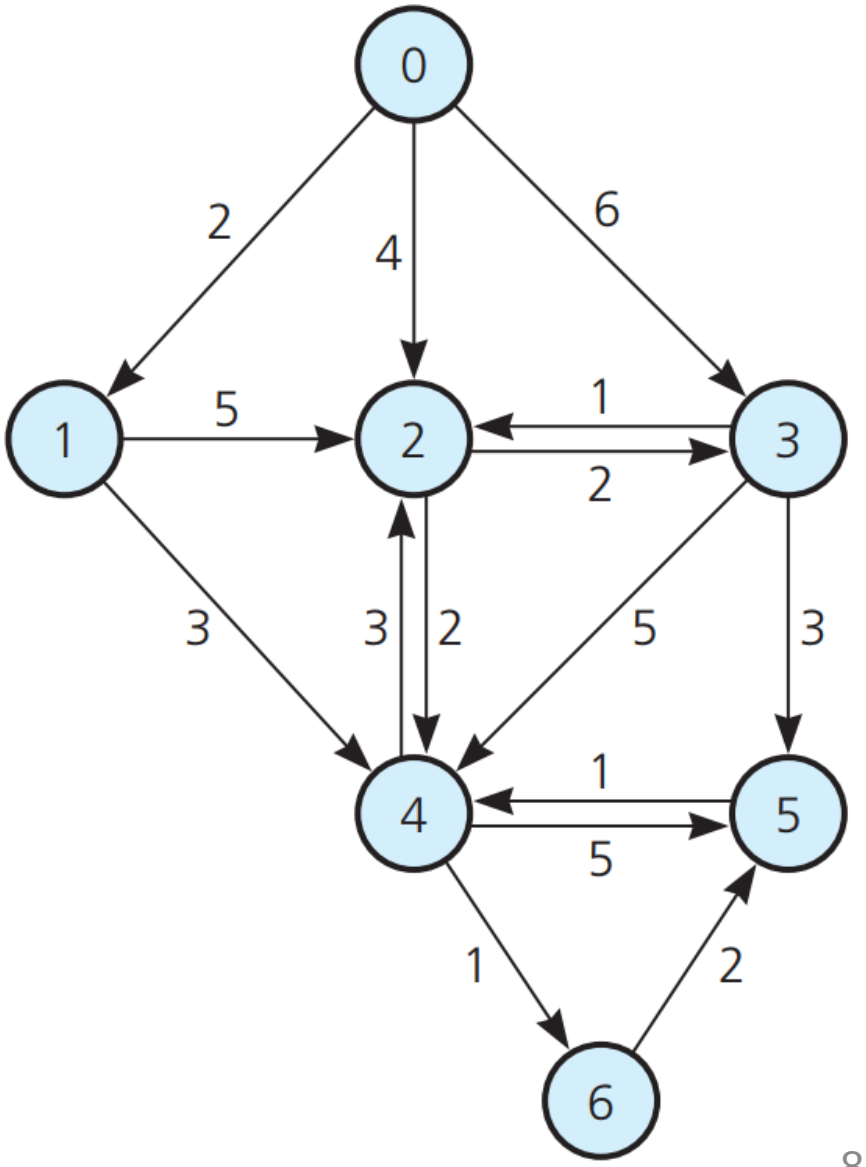




			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

Exercies

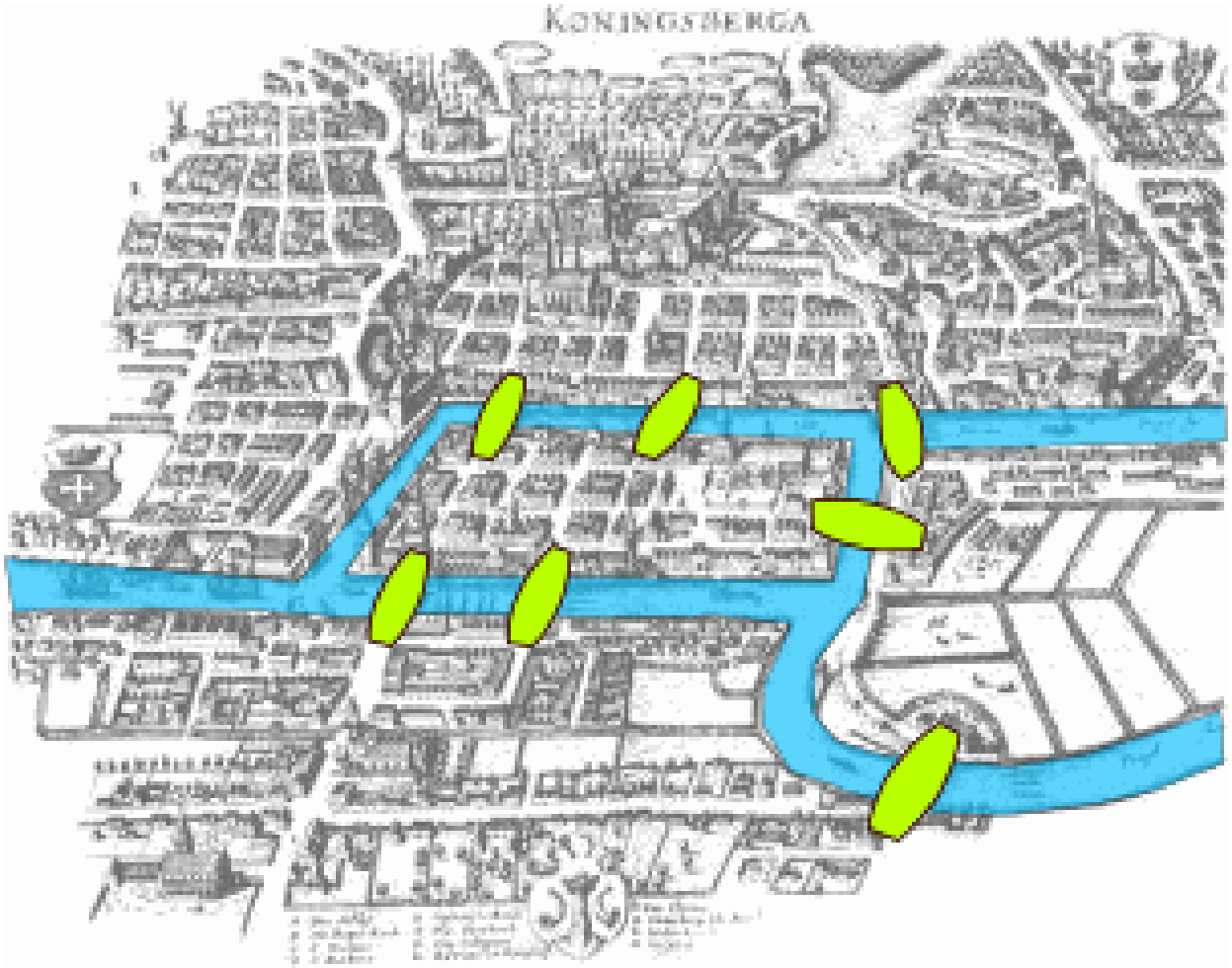
- Trace the shortest-path algorithm for the graph in following figure, letting vertex 0 be the origin.



Applications of graphs 5

The city of Königsberg in Prussia (now Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands - Kneiphof and Lomse - which were connected to each other, or to the two mainland portions of the city, by seven bridges.

The problem was to devise a walk through the city that would cross each of those bridges once and only once.



https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

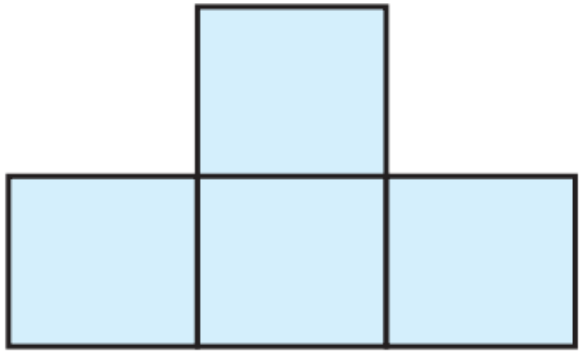
Euler's Analysis

Euler demonstrated that no solution exists for this particular configuration of edges and vertices.

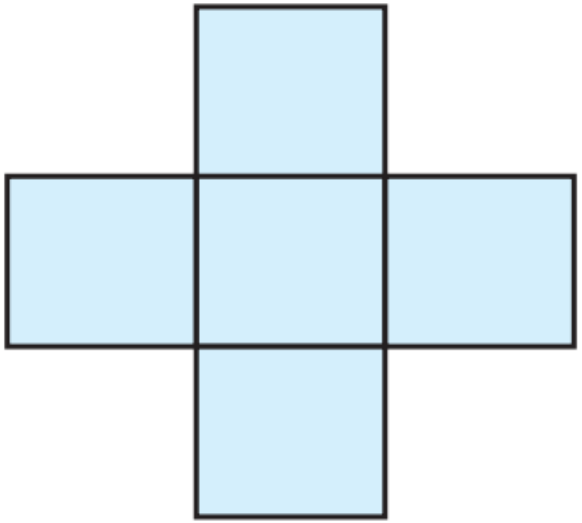
Pencil and paper drawings

- Let drawing each of the diagrams in the following figure without lifting your pencil or redrawing a line, and ending at your starting point.

(a)



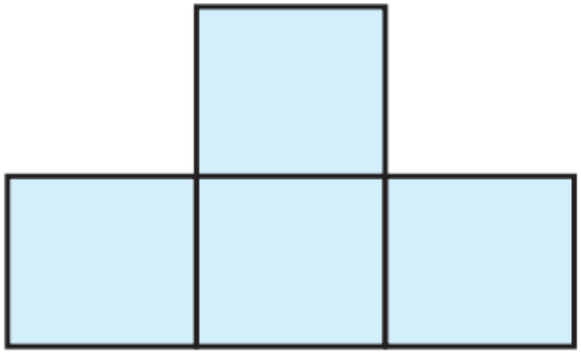
(b)



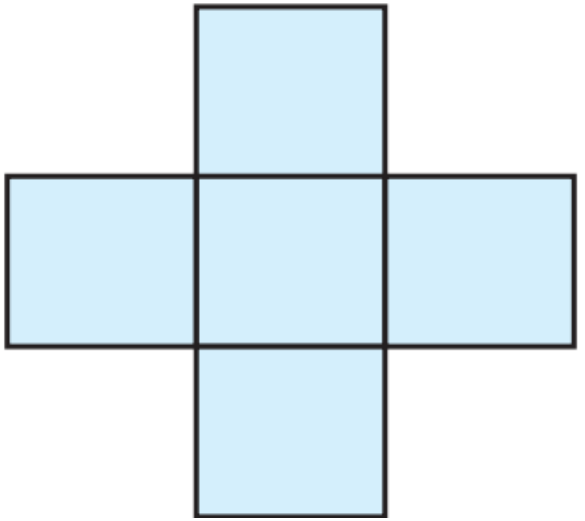
Pencil and paper drawings

Graph viewing

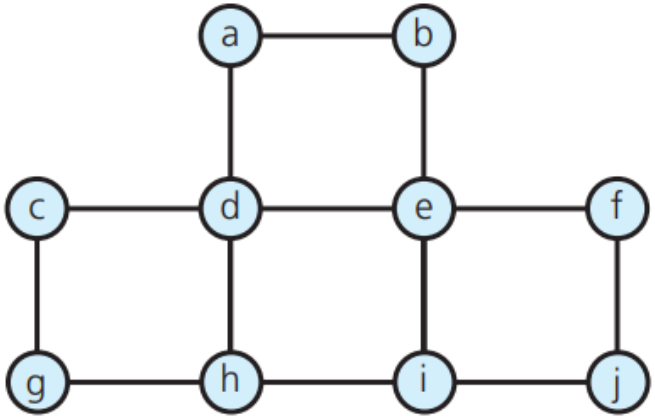
(a)



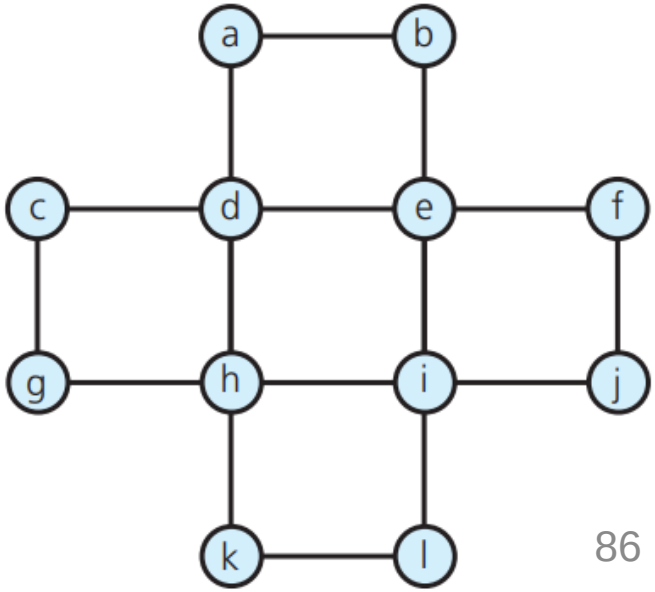
(b)



(a)



(b)



Euler's Analysis

- A path in an undirected graph that begins at a vertex v, passes through every edge in the graph exactly once, and terminates at v is called an Euler circuit.
- An Euler circuit exists if and only if each vertex touches an even number of edges.

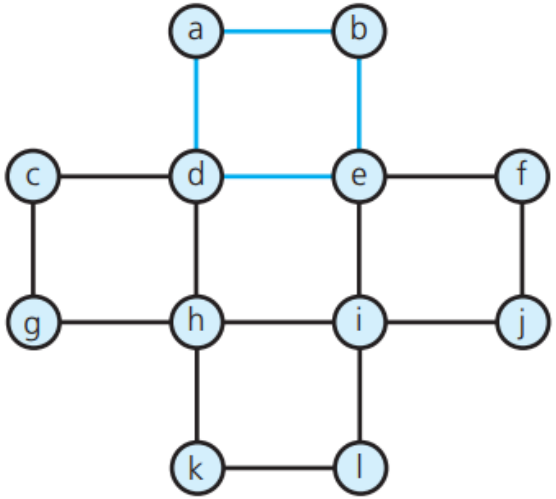
DFS for finding Euler Circuit

- The strategy uses a depth-first search that marks edges instead of vertices as they are traversed.

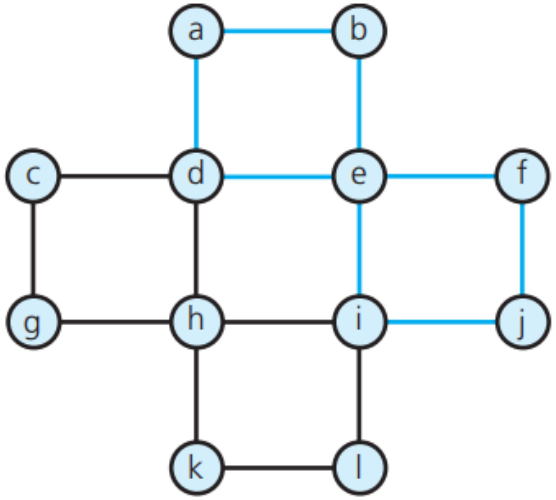
```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Recursive version.
dfs(v: Vertex)
      for (each unvisited vertex u adjacent to v)
            Mark edge (v,u) as visited
            dfs(u)
```

DFS for finding Euler Circuit

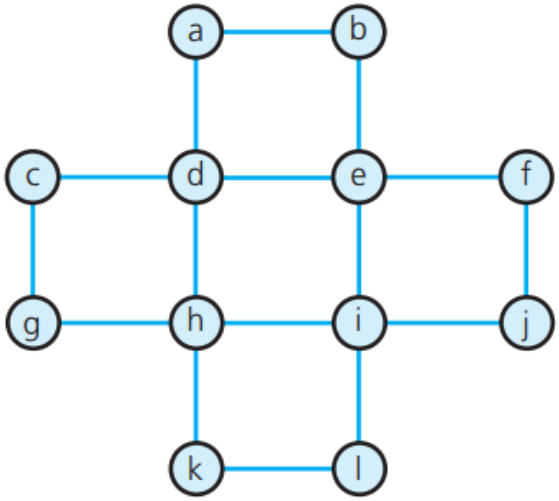
(a)



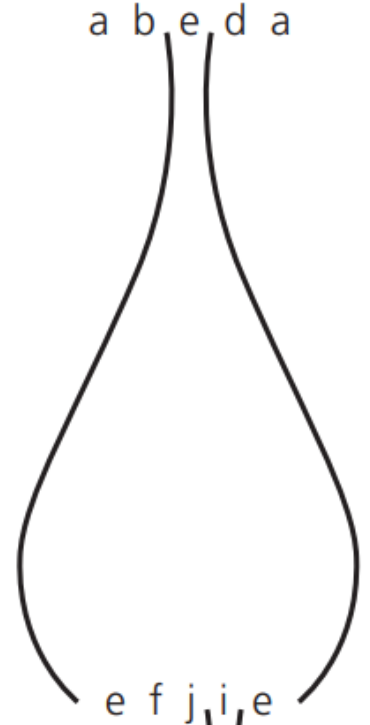
(b)

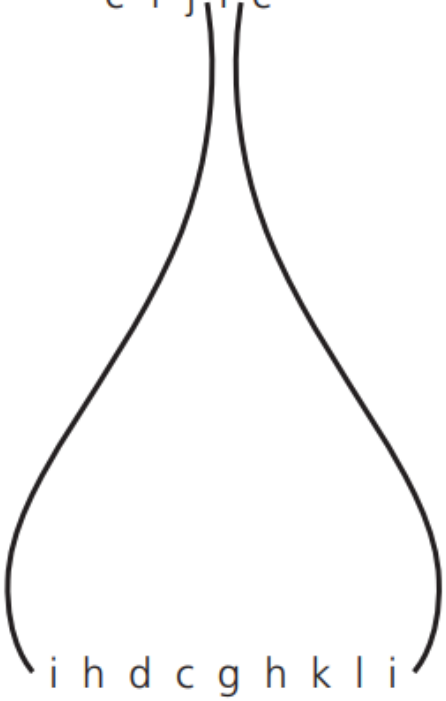


(c)



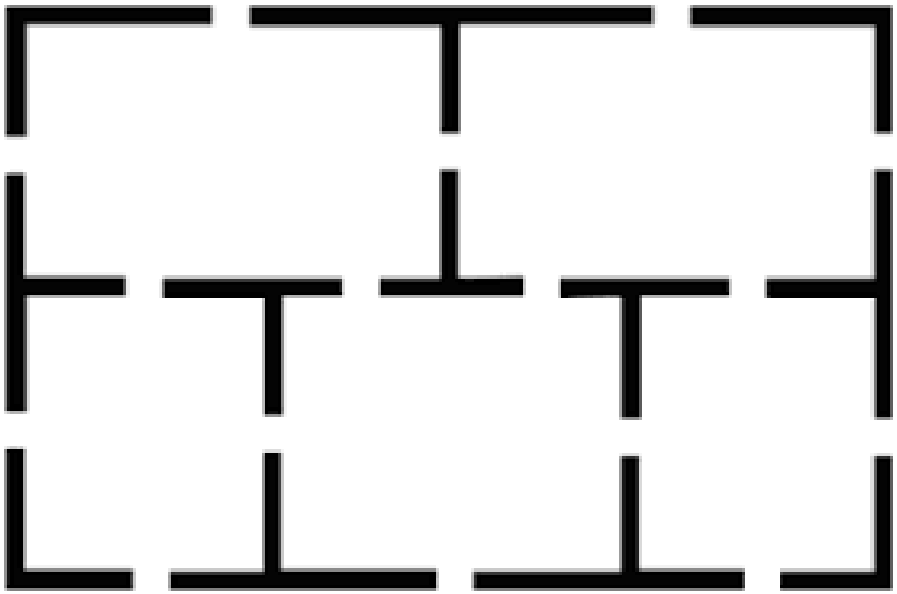
Euler circuit: a b e f j i h d c g h k l i e d a

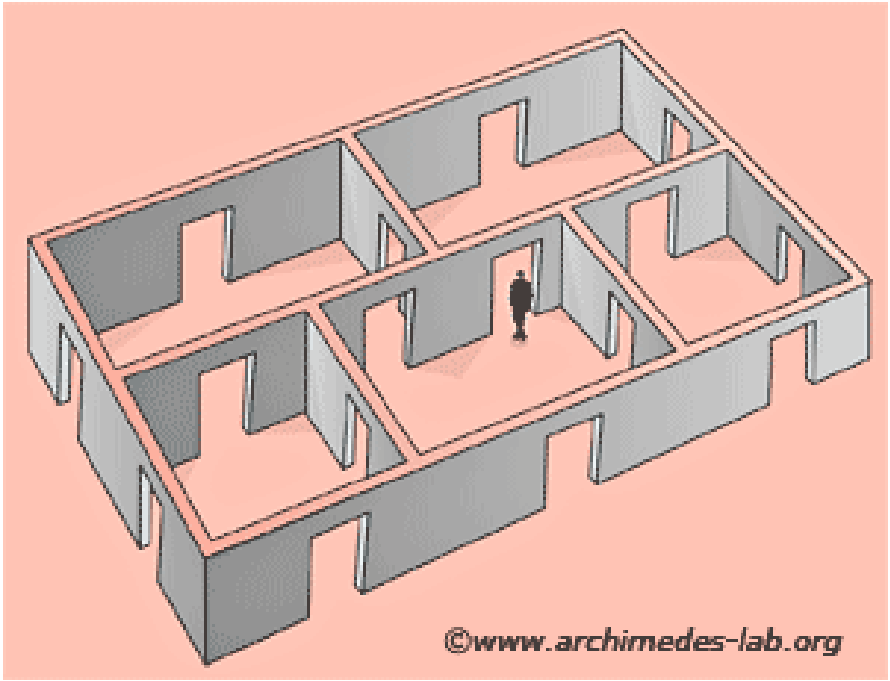




Five room puzzle

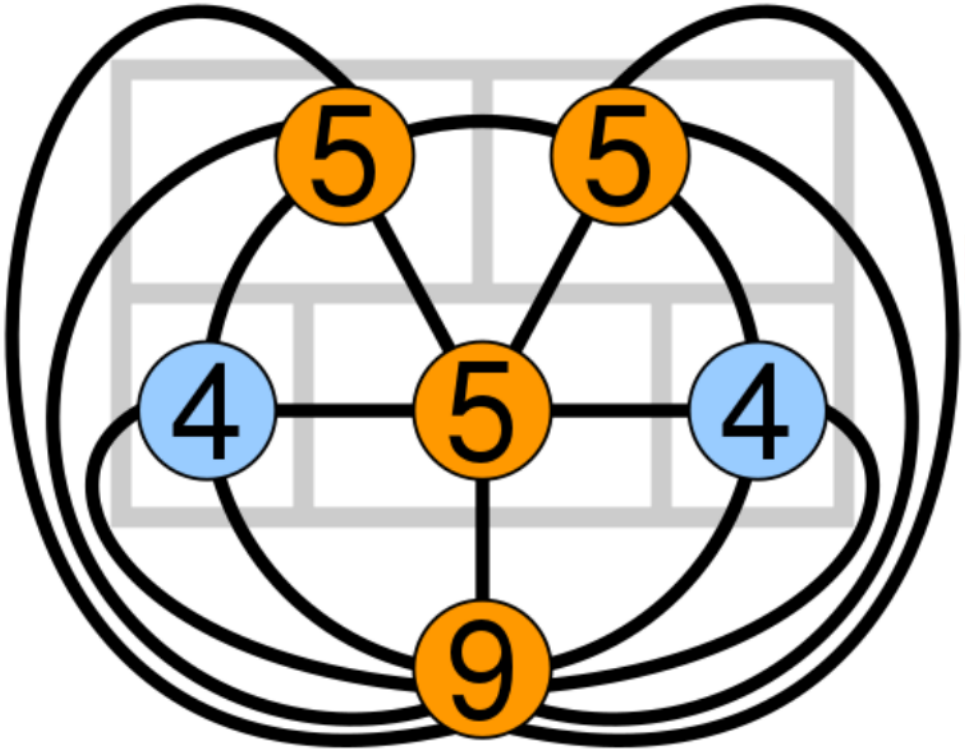
- A large rectangle divided into five "rooms". The objective of the puzzle is to cross each "wall" of the diagram with a continuous line only once.

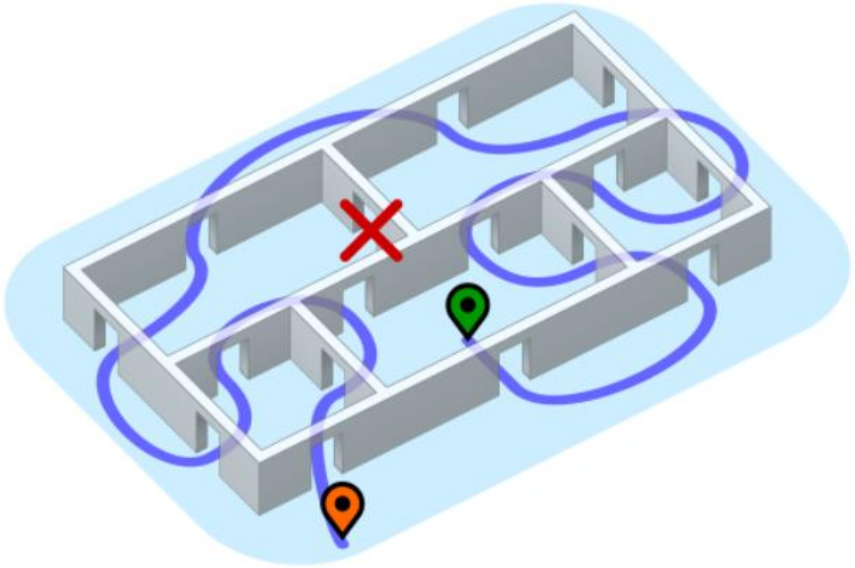




Five room puzzle

- The puzzle may be represented in graphical form with each room corresponding to a vertex (including the outside area as a room) and two vertices joined by an edge if the rooms have a common wall.

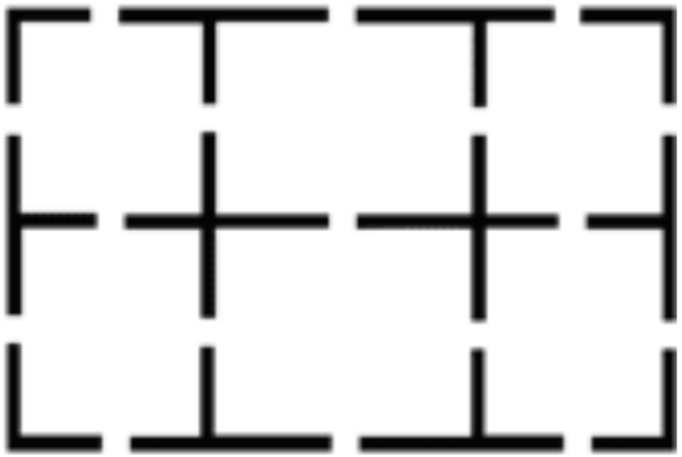




This puzzle cannot be solved

Out-of-the box Solution

- If we close a door or add an extra room to the puzzle, then it becomes solvable.
- Or by permitting passage through more than one wall at a time (that is, through a corner of a room), or by solving the puzzle on a torus (doughnut) instead of a flat plane



Content

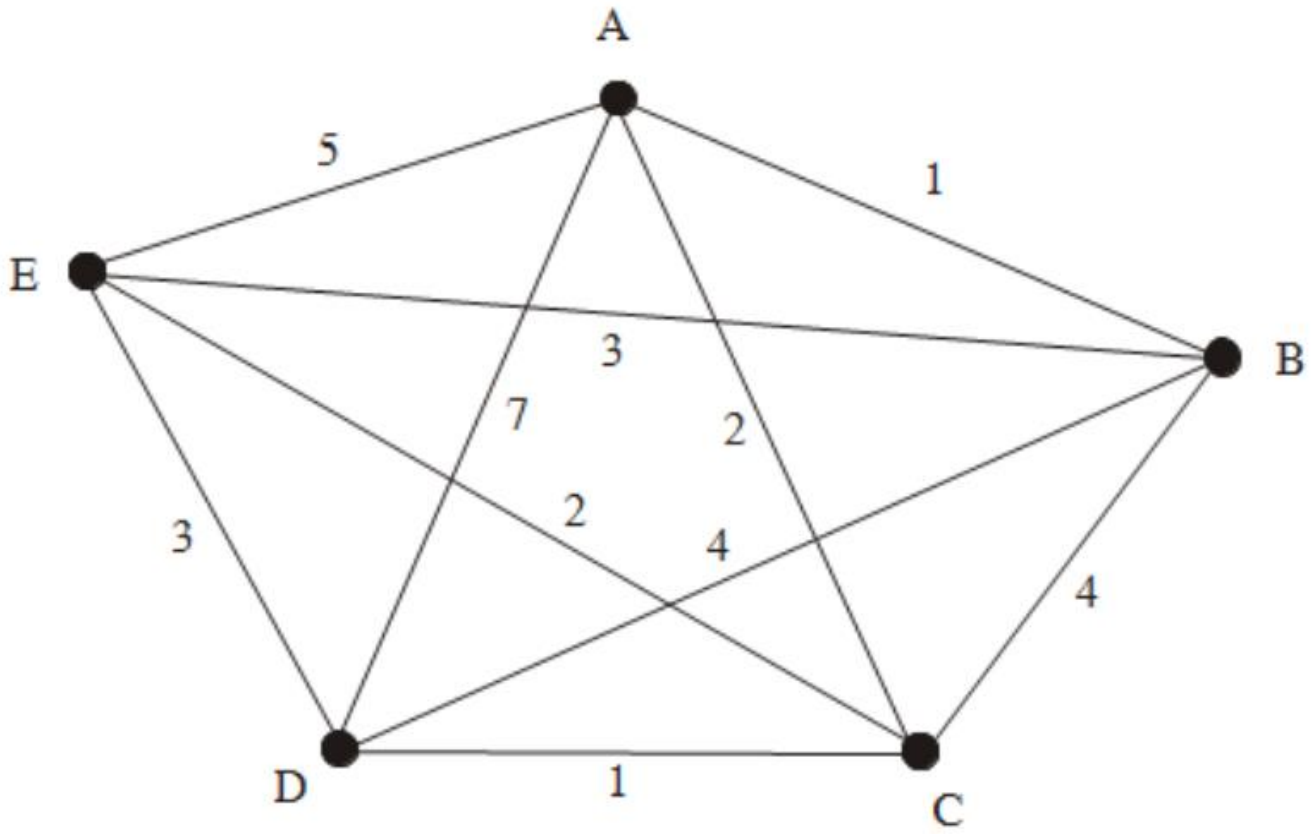
- Introduction
- Basic Concepts
- Graph Abstract Data Type
- Graph Traversal
- Graph Applications
- Some Difficult Problems

Some Difficult Problems - TSP

- A Hamilton circuit is a path that begins at a vertex v, passes through every vertex exactly once, and terminates at v.
  - Determining whether an arbitrary graph contains a Hamilton circuit can be difficult.
- A well-known variation of this problem is the traveling salesperson problem.
  - The salesperson must begin at an origin city, visit every other city exactly once, and return to the origin city. However, the circuit traveled must be the least expensive.

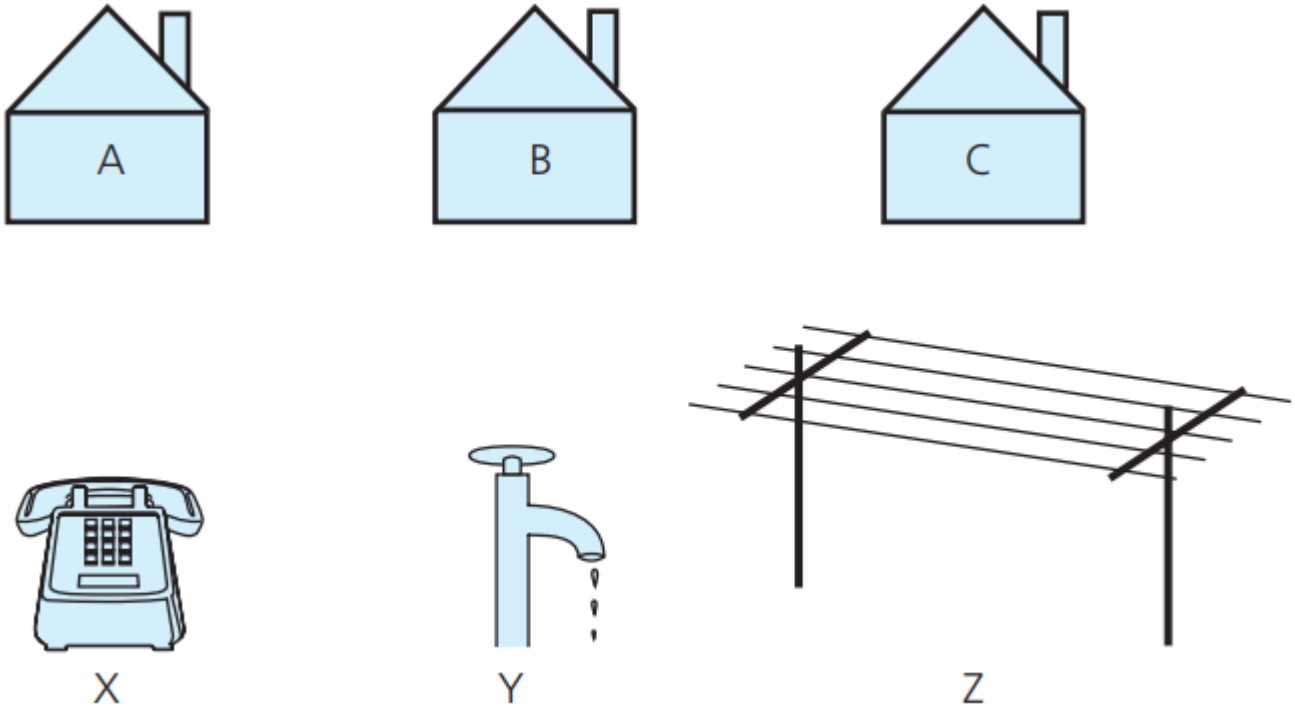
Some Difficult Problems - TSP

- An TSP example



Three Utilities Problem

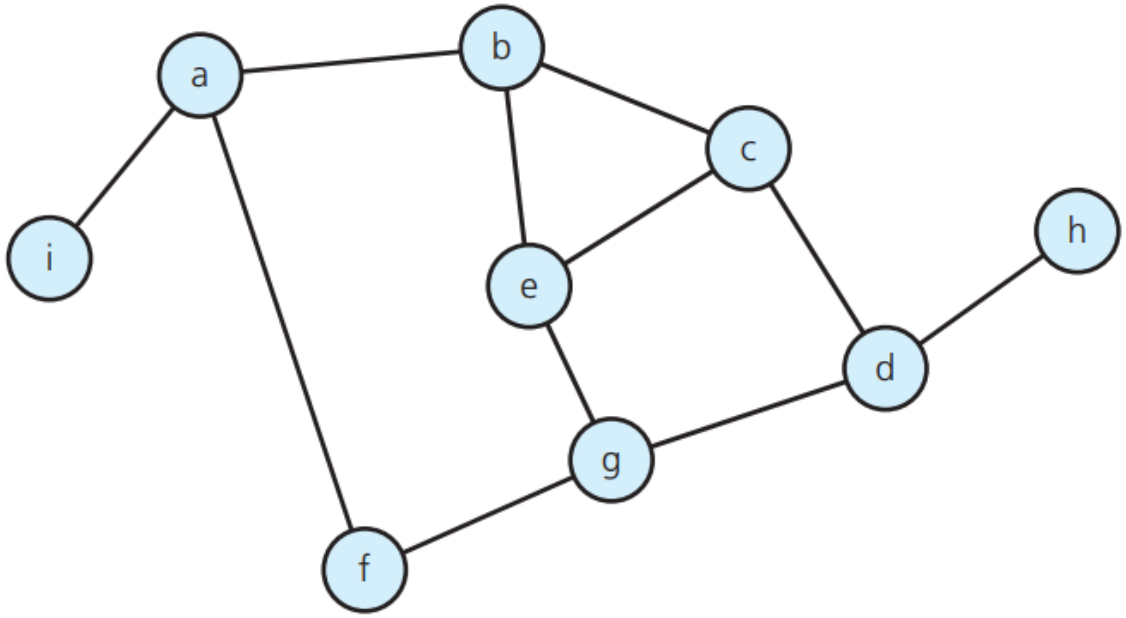
- Imagine three houses A, B, and C and three utilities X, Y, and Z (such as telephone, water, and electricity).
  - If the houses and the utilities are vertices in a graph, is it possible to connect each house to each utility with edges that do not cross one another?



The answer to this question is no.

The Four-Color Problem

- Given a planar graph, can you color the vertices so that no adjacent vertices have the same color? How many color at least to do that?



References

- Frank M. Carrano, Data Abstraction and Problem Solving with C++: Walls and Mirrors, Frank M. Carrano, 6th Edition
- Minimum Spanning Tree in Graph - Week 13. 2 Problem: Laying Telephone Wire Central office.

The End.