FPT UNIVERSITY
DREAM OF INNOVATION

5. Graphs

Part 1

Objectives

- Graph Introduction
- Graph definition
- Graph Terminology
- Graph applications
- Graph Representation
- Graph Traversals
- Shortest Paths
  - Dijsktra algorithm
  - Floyd algorithm

Graph Introduction

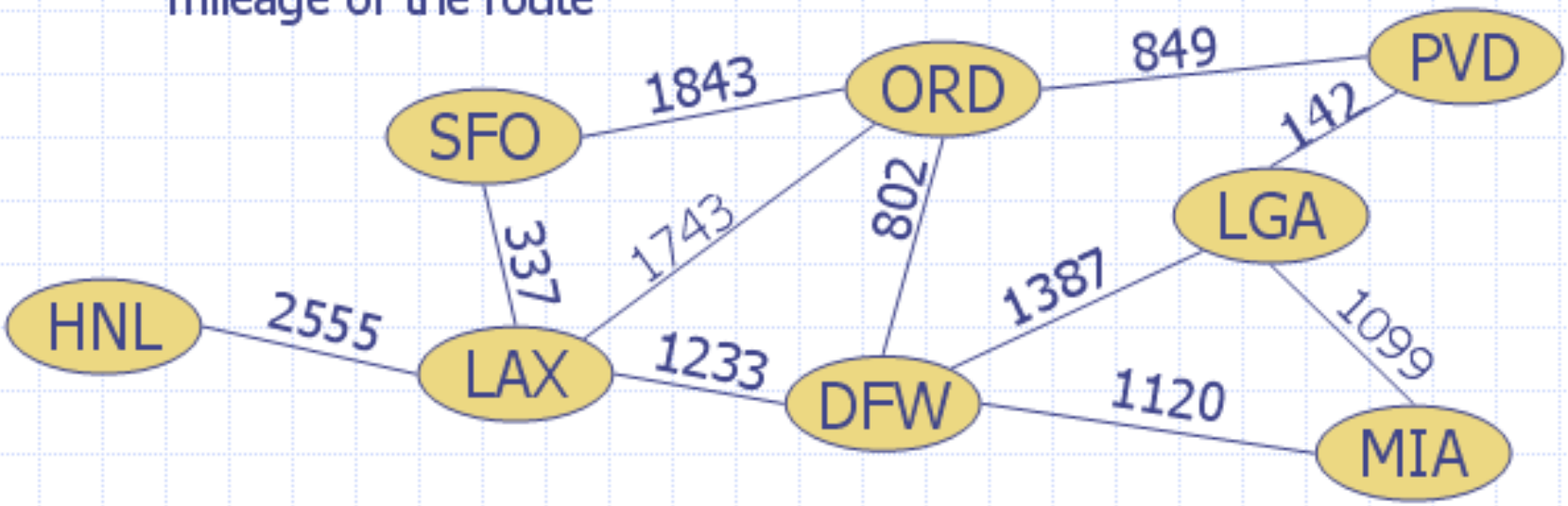
A ***graph*** is a way of representing relationships that exist between pairs of objects. That is, a graph is a **set of** objects, called **vertices**, together with a collection of pairwise connections between them, called **edges**. Graphs have applications in modeling many domains, including mapping, transportation, computer networks, and electrical engineering. By the way, this notion of a “graph” should not be confused with bar charts and function plots, as these kinds of “graphs” are unrelated to the topic of this chapter.

Graph definition

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges

In briefly saying, a **graph** is a collection of vertices and the connections between them

- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



Graph Terminology - 1

- Edges in a graph are either ***directed*** or ***undirected***. An edge (*u*,*v*) is said to be ***directed*** from *u* to *v* if the pair (*u*,*v*) is ordered, with *u* preceding *v*. An edge (*u*,*v*) is said to be ***undirected*** if the pair (*u*,*v*) is not ordered. Undirected edges are sometimes denoted with set notation, as {*u*,*v*}.
- If all the edges in a graph are undirected, then we say the graph is an ***undirected graph***. Likewise, a ***directed graph***, also called a ***digraph***, is a graph whose edges are all directed. A graph that has both directed and undirected edges is often called a ***mixed graph***.

Note that an undirected or mixed graph can be converted into a directed graph by replacing every undirected edge (*u*,*v*) by the pair of directed edges (*u*,*v*) and (*v*,*u*). It is often useful, however, to keep undirected and mixed graphs represented as they are, for such graphs have several applications.

Graph Examples - 1

- A **city map** can be modeled as a **graph** whose vertices are intersections or dead ends, and whose edges are stretches of streets without intersections. This graph has both undirected edges, which correspond to stretches of two-way streets, and directed edges, which correspond to stretches of one-way streets. Thus, in this way, a graph modeling a city map is a **mixed graph**.
- Physical examples of graphs are present in the **electrical wiring** and **plumbing networks** of a building. Such networks can be modeled as graphs, where each connector, fixture, or outlet is viewed as a vertex, and each uninterrupted stretch of wire or pipe is viewed as an edge. Such graphs are actually components of much larger graphs, namely the local power and water distribution networks.

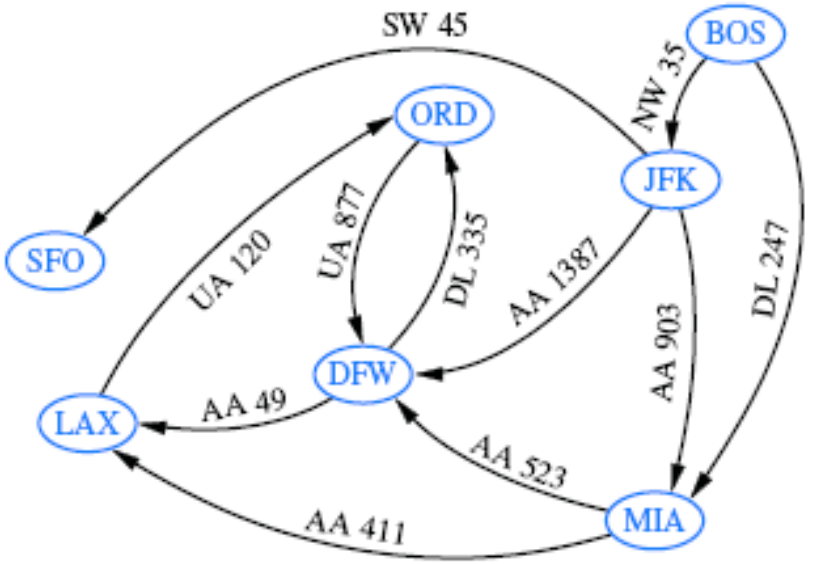
Graph Terminology - 2

- The two vertices joined by an edge are called the ***end vertices*** (or ***endpoints***) of the edge. If an edge is directed, its first endpoint is its ***origin*** and the other is the ***destination*** of the edge. Two vertices *u* and *v* are said to be ***adjacent*** if there is an edge whose end vertices are *u* and *v*. An edge is said to be ***incident*** to a vertex if the vertex is one of the edge's endpoints. The ***outgoing edges*** of a vertex are the directed edges whose origin is that vertex. The ***incoming edges*** of a vertex are the directed edges whose destination is that vertex. The ***degree*** of a vertex *v*, denoted **deg(*v*)**, is the number of incident edges of *v*. If deg(u) = 0 then u is called ***isolated vertex***. The ***in-degree*** and ***out-degree*** of a vertex *v* are the number of the incoming and outgoing edges of *v*, and are denoted indeg(*v*) and outdeg(*v*), respectively.

Graph Examples - 2

- We can study air transportation by constructing a graph *G*, called a **flight network**, whose vertices are associated with airports, and whose edges are associated with flights. In graph *G*, the edges are directed because a given flight has a specific travel direction. The endpoints of an edge *e* in *G* correspond respectively to the origin and destination of the flight corresponding to *e*.

  Two airports are adjacent in *G* if there is a flight that flies between them, and an edge *e* is incident to a vertex *v* in *G* if the flight for *e* flies to or from the airport for *v*.



Graph Terminology - 3

- The definition of a graph refers to the group of edges as a ***collection***, not a ***set***, thus allowing two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called ***parallel edges*** or ***multiple edges***. A flight network can contain parallel edges. A graph containing multiple edges but no loops is called **multigraph**.
- Another special type of edge is one that connects a vertex to itself. Namely, we say that an edge (undirected or directed) is a ***self-loop*** if its two endpoints coincide. A self-loop may occur in a graph associated with a city map, where it would correspond to a “circle” (a curving street that returns to its starting point).
- With few exceptions, graphs do not have parallel edges or self-loops. Such graphs are said to be ***simple***. Thus, we can usually say that the edges of a simple graph are a ***set*** of vertex pairs (and not just a collection). A graph is considered simple unless otherwise specified.

Graph Terminology - 4

- A ***path*** is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex. A ***cycle*** is a path that starts and ends at the same vertex, and that includes at least one edge. We say that a path is ***simple*** if each vertex in the path is distinct, and we say that a cycle is ***simple*** if each vertex in the cycle is distinct, except for the first and last one. A ***directed path*** is a path such that all edges are directed and are traversed along their direction. A ***directed cycle*** is similarly defined.
- If a graph is simple, we may omit the edges when describing path *P* or cycle *C*, as these are well defined, in which case *P* is a list of adjacent vertices and *C* is a cycle of adjacent vertices.

Graph Terminology - 5

- A undirected graph G is ***connected*** if, for any two vertices, there is a path.
- A directed graph *G* is ***strongly connected*** if for any two vertices u and v, there is a directed path. A directed graph is called **weakly connected** if replacing all of its directed edges with undirected edges produces a connected (undirected) graph.
- A ***subgraph*** of a graph *G* is a graph *H* whose vertices and edges are subsets of the vertices and edges of *G*, respectively. A ***spanning subgraph*** of *G* is a subgraph of *G* that contains all the vertices of the graph *G*. If a graph *G* is not connected,
- its maximal connected subgraphs are called the ***connected components*** of *G*. A ***forest*** is a graph without cycles. A ***tree*** is a connected forest, that is, a connected graph without cycles. A ***spanning tree*** of a graph is a spanning subgraph that is a tree. (Note that this definition of a tree is somewhat different from the one given in Chapter 4, as there is not necessarily a designated root.)

Graph Terminology - 6

- If the vertex is removed from a graph (along with incident edges) and there is no way to find a path from *a* to *b,* then the graph is split into two separate subgraphs called **articulation points**, or **cut-vertices**
- If an edge causes a graph to be split into two subgraphs, it is called a **bridge** or **cut-edge**
- Connected subgraphs with no articulation points or bridges are called **blocks**

We can use depth first traverse to check the connectivity of a graph

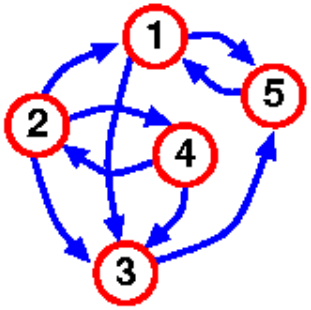
Graph Examples - 3

- Perhaps the most talked about graph today is the **Internet**, which can be viewed as a graph whose vertices are computers and whose (undirected) edges are communication connections between pairs of computers on the Internet. The computers and the connections between them in a single domain, like wiley.com, form a subgraph of the Internet. If this subgraph is connected, then two users on computers in this domain can send email to one another without having their information packets ever leave their domain. Suppose the edges of this subgraph form a spanning tree. This implies that, if even a single connection goes down (for example, because someone pulls a communication cable out of the back of a computer in this domain), then this subgraph will no longer be connected.

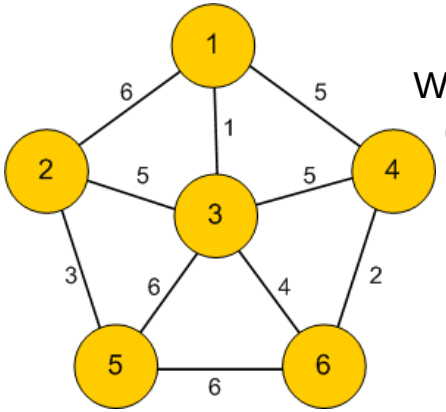
Graph Examples - 4

Single undirected graph

It is also a simple graph because it has no mutiple edges and no loops

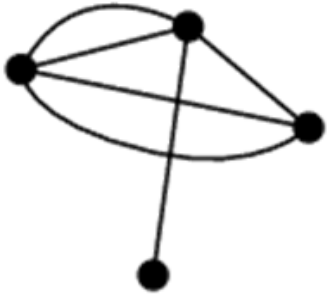


Single Directed graph

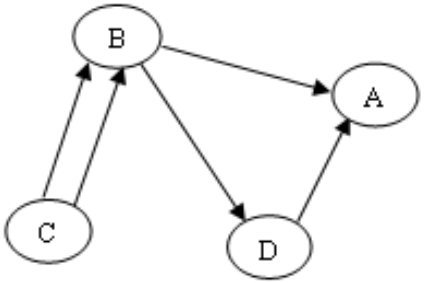


Weighted graph

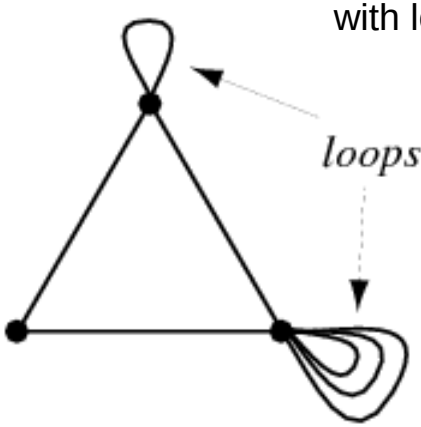
An undirected graph with multiple edges



An directed graph with multiple edges

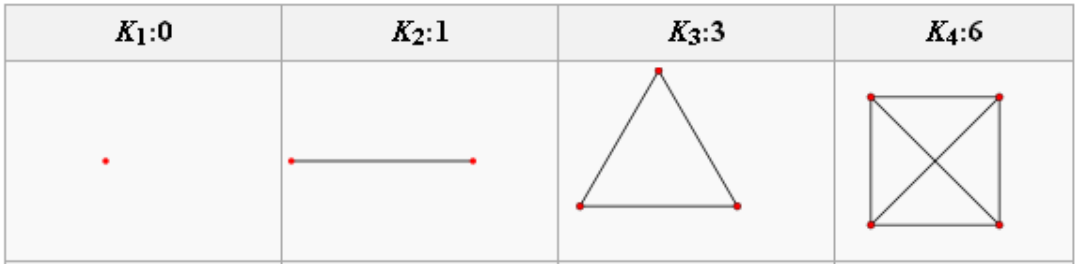


An undirected graph with loops



Graph Terminology - 7

- A **complete graph** is a graph where every pair of vertices is connected by an edge.
- A **simple complete graph on *n* vertices** has *n* vertices and *n*(*n*-1)/2 edges, and is denoted by *Kn*



- A *graph* G'=(V', E') is a subgraph of another graph G=(V, E) iff V

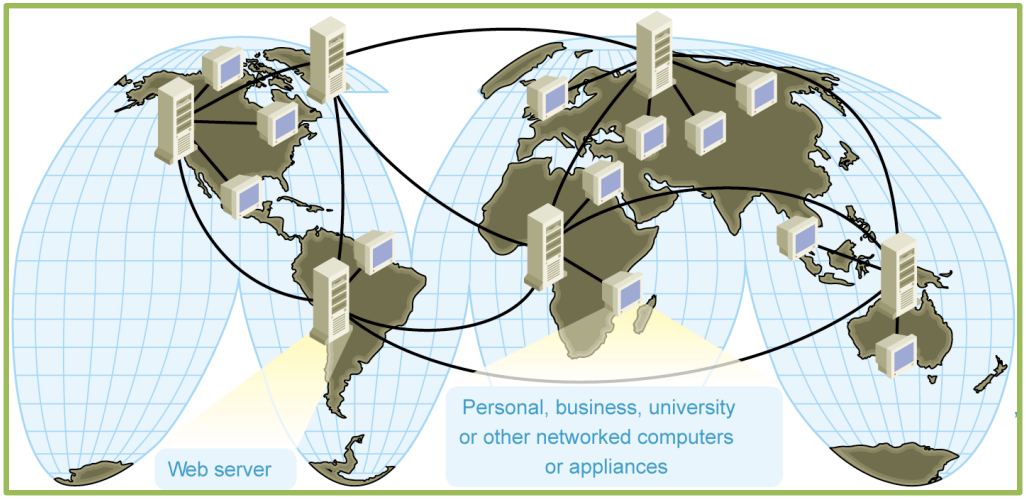
$$V' \subseteq V \text{ and } E' \subseteq E$$

Summary for Basic Notions on Graph

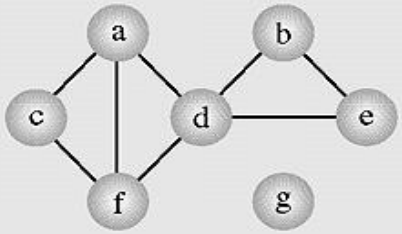
Vertex, edge, (directed, undirected edge), (directed, undirected graph), adjacent vertices, incident edge,degree, isolated vertex, parallel edge or multiple edge, loop, simple graph, multigraph, path, simple path, cycle, connected, strongly connected, weakly connected, subgraph, spanning subgraph, forest, tree, spanning tree, complete graph, simple complete graph, cut-vertex (articulation point), bridge(cut-edge), block,...

Graph applications

- Electronic circuits
- Transportation networks
- Computer networks
- Database
  - Entity-relationship diagram



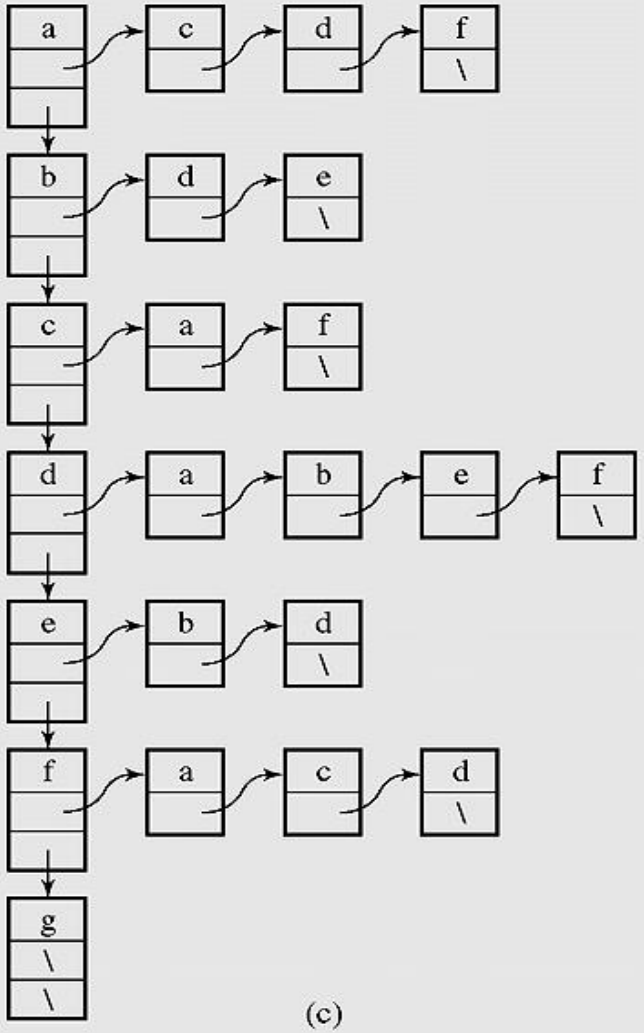
Graph Representation – 1
(Adjacency list)



(a)

a	c	d	f	
b	d	e		
c	a	f		
d	a	b	e	f
e	b	d		
f	a	c	d	
g				

(b)



(c)

Graph representations.
A graph (a) can be **represented as (b–c) an adjacency list.**

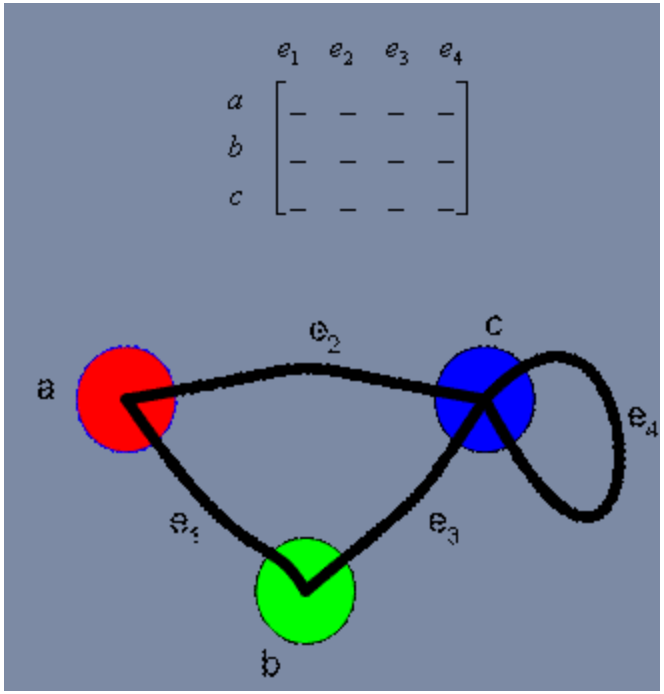
Graph Representation – 2
(Adjacency matrix)

	a	b	c	d	e	f	g
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

Graph represented by an adjacency matrix

Graph Representation – 3
(Incident matrix)

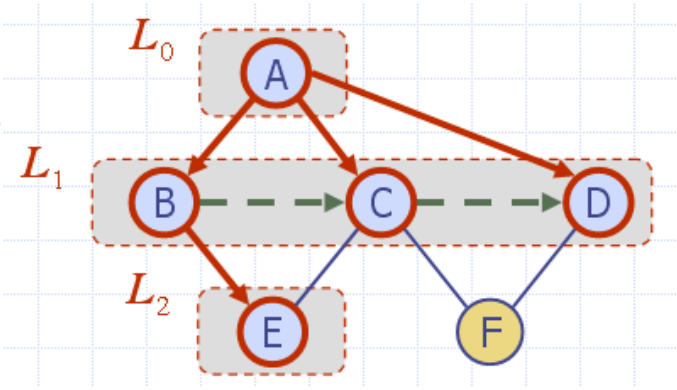
A vertex is said to be *incident* to an edge if the edge is connected to the vertex.

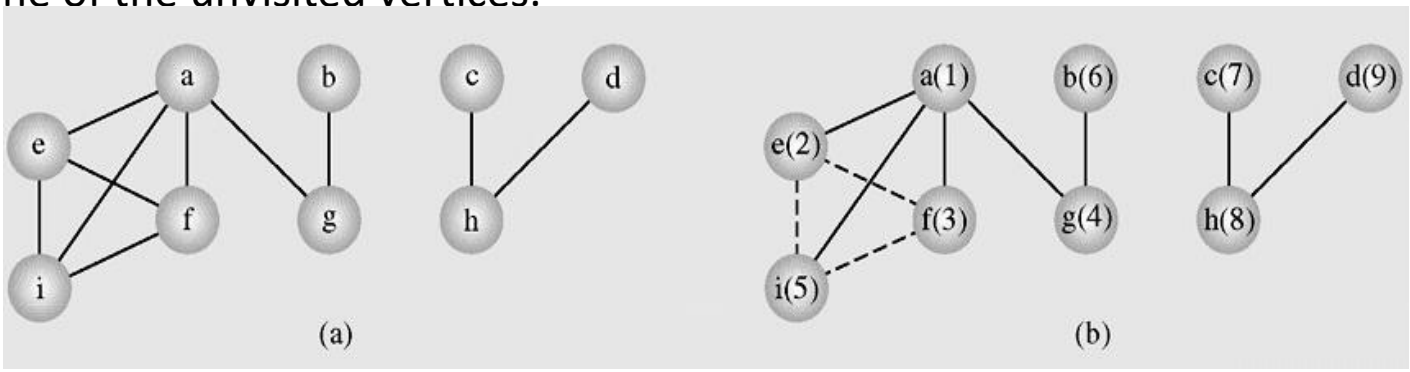


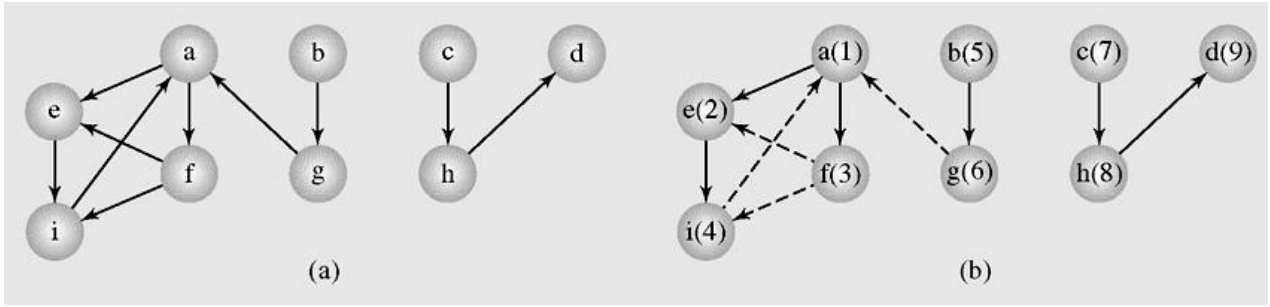
Graph represented by incident matrix

Graph Traversals
Breadth-first Search

- In the **breadth-first search algorithm**, at first a selected vertex *v* is visited and then each unvisited vertex adjacent to *v* is visited. Suppose these adjacents are v_1 , v_2,… v_k. After all these vertices are visited, all adjacents of v_1 are visited, then all adjacents of v_2 are visited and so on. If there are still some unvisited vertices in the graph, the traversal continues restarting for one of the unvisited vertices.







Breadth-first Search Algorithm

Search a graph (directed or not) in breadth first, this is done by using a queue where the vertices found are stored.

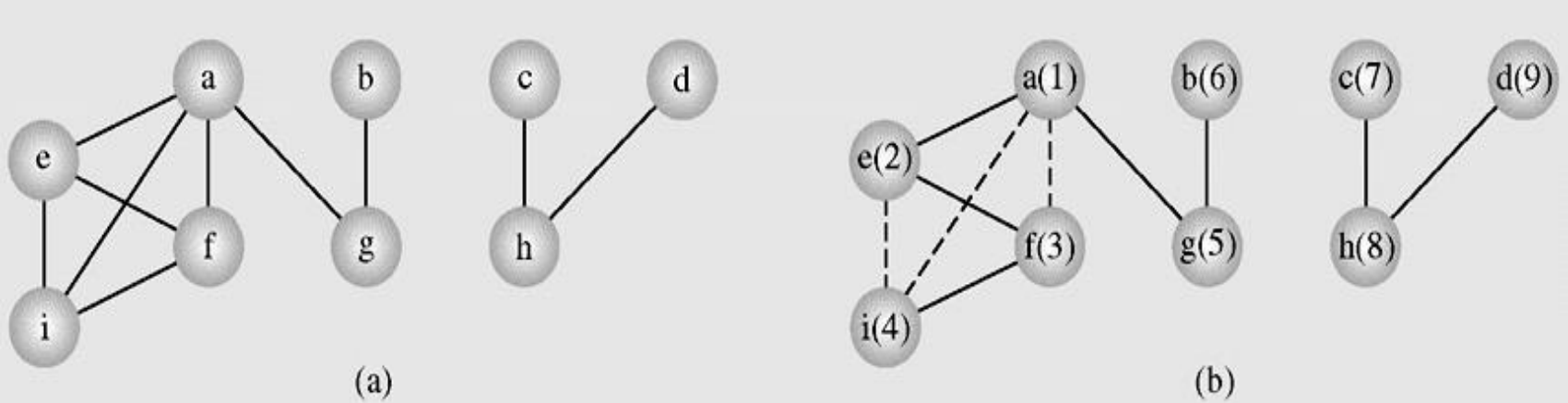
```
BFS(Graph G)
  { all vertices of G are first painted white
    the graph root is painted gray and put in a queue
    while the queue is not empty
       {  a vertex u is removed from the queue
          for all white successors v of u
              { v is painted gray
                v is added to the queue
              }
          u is painted black
       }
  }
```

Breadth-first Search code

```
// bread first traverse from vertex k
void breadthFirst(int k)
  {MyQueue q = new MyQueue();int i,h;
    boolean [] enqueued = new boolean[n];
    for(i=0;i<n;i++) enqueued[i]=false;
    q.enqueue(new Integer(k));enqueued[k]=true;
    while(!q.isEmpty())
     {h=Integer.parseInt((q.dequeue()).toString().trim());
      visit(h);
      for(i=0;i<n;i++)
         if((!enqueued[i]) && a[h][i]>0)
           {q.enqueue(new Integer(i));
            enqueued[i] = true;
           }
     }
    System.out.println();
   }
```

Depth-first Traversal

- In the **depth-first search algorithm**, at first selected vertex *v* is visited and then each unvisited vertex adjacent to *v* is visited by depth-first search. If there are still some unvisited vertices in the graph, the traversal continues restarting for one of the unvisited vertices.



An example of application of the `depthFirstSearch()` algorithm to a graph

Depth-First Search algorithm

```
DFS-visit (Graph G, Vertex u)
   { the vertex u is painted gray
     for all white successors v of u
        { dfs-visit(G, v)
        }
     u is painted black
   }
DFS (Graph G)
   { all vertices of G are first painted white
     DFS-visit(G, root of G)
   }
```

The idea is the same, but we now use a stack instead of a queue. With recursion of course, so the stack management is all done by Java.
Here is a brief description of the DFS algorithm:

DFS on a graph with ***n*** vertices and ***m*** edges takes ***O***(***n*** + ***m***) time

Depth-First Traverse code

```
void visit(int i)
  {System.out.print(" " + v[i]);
  }
void depthFirst(boolean visited[], int i)
  {visit(i);visited[i] = true;
   int j;
   for(j=0;j<n;j++)
       if(a[i][j]>0 && (!visited[j]))
          depthFirst(visited,j);
  }
 void depthFirst(int k)
  {int i; boolean [] visited = new boolean[20];
   for(i=0;i<n;i++) visited[i]=false;
   depthFirst(visited,k);
   for(i=0;i<n;i++)
    if(!visited[i])
      depthFirst(visited,i);
   System.out.println();
  }
```

Shortest Path problem

- *The problem:* find the shortest path between a pair of vertices of a graph
- *The graph*: may contain negative edges but no negative cycles
- *A representation*: a weighted matrix where
  W(i,j) = 0 if i=j.
  W(i,j) = ∞ if there is no edge between i and j.
  W(i,j) = weight of the edge (i,j)

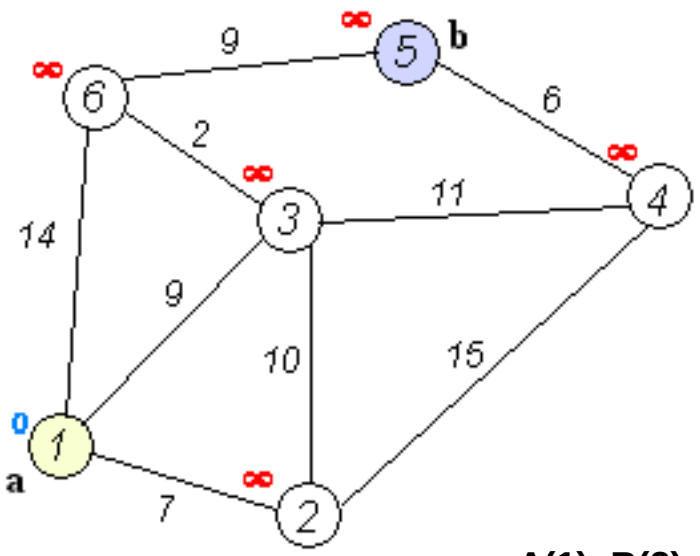
Dijkstra's Algorithm - 1

The main idea in applying the greedy-method pattern to the single-source shortestpath problem is to perform a “weighted” breadth-first search starting at the source vertex *s*. In particular, we can use the greedy method to develop an algorithm that iteratively grows a “cloud” of vertices out of *s*, with the vertices entering the cloud in order of their distances from *s*. Thus, in each iteration, the next vertex chosen is the vertex outside the cloud that is closest to *s*. The algorithm terminates when no more vertices are outside the cloud (or when those outside the cloud are not connected to those within the cloud), at which point we have a shortest path from *s* to every vertex of *G* that is reachable from *s*. This approach is a simple, but nevertheless powerful, example of the greedy-method design pattern. Applying the greedy method to the single-source, shortest-path problem, results in an algorithm known as ***Dijkstra’s algorithm***.

Dijkstra's Algorithm - 2

```
DijkstraAlgorithm(non-gegative weighted simple
digraph, vertex first)
for all vertices  v # first currDist(v) = ∞;
currDist(first) = 0;
toBeChecked = V (all vertices);
checked = empty;
while toBeChecked is not empty
 u = a vertex in toBeChecked with min.currDist(u);
 remove u from toBeChecked and add to checked;
 for all vertices v adjacent to u and in toBeChecked
  if (currDist(v) > currDist(u) + weight(edge(uv)))
    {currDist(v) = currDist(u) + weight(edge(uv))
     predeccessor(v) = u;
    }
```

Dijkstra's Algorithm example - 1




```
Dijkstra algorithm for shortest path from A to  E:
     The S set:          B         C         D         E         F

 0:            A  (  7,A)  (  9,A)  (INF,A)  (INF,A)  ( 14,A)

 1:           AB  (  7,A)  (  9,A)  ( 22,B)  (INF,A)  ( 14,A)

 2:          ABC           (  9,A)  ( 20,C)  (INF,A)  ( 11,C)

 3:         ABCF                    ( 20,C)  ( 20,F)  ( 11,C)

 4:        ABCFD                    ( 20,C)  ( 20,F)

 5:       ABCFDE                             ( 20,F)

The length of shortest path from A to  E is  20

Path:
A -> C -> F -> E
```

A(1), B(2), C(3), D(4), E(5), F(6)

Dijkstra's algorithm keeps *two* sets of vertices:

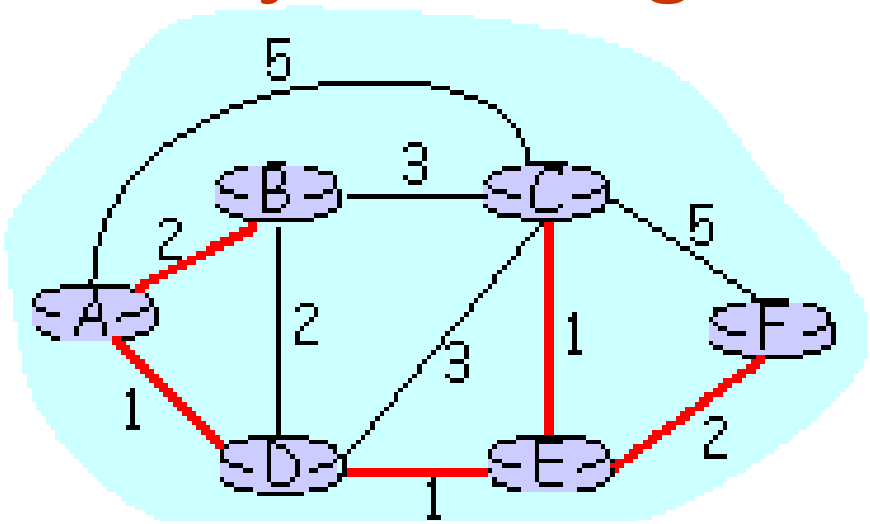
- S — Vertices whose shortest paths have already been determined
- V-S — Remainder

Also

- d — Best estimates of shortest path to each vertex
- p — Predecessors for each vertex

```
int [][] b = {
   {  0,   7,   9, 99, 99, 14},
   {  7,   0, 10, 15, 99, 99},
   {  9, 10,   0, 11, 99,   2},
   {99, 15, 11,   0,   6, 99},
   {99, 99, 99,   6,   0,   9},
   {14, 99,   2, 99,   9,   0}
  };
```

Dijkstra's Algorithm example - 2



The complexity of Dijkstra's algorithm is $O(|V|^2)$. This algorithm is not general enough in that it may fail when negative weights are used in graphs.

Dijkstra's algorithm: example

Step	start N	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	infinity	infinity
1	AD	2,A	4,D		2,D	infinity
2	ADE	2,A	3,E			4,E
3	ADEB		3,E			4,E
4	ADEBC					4,E
5	ADEBCF					

Floyd Algorithm

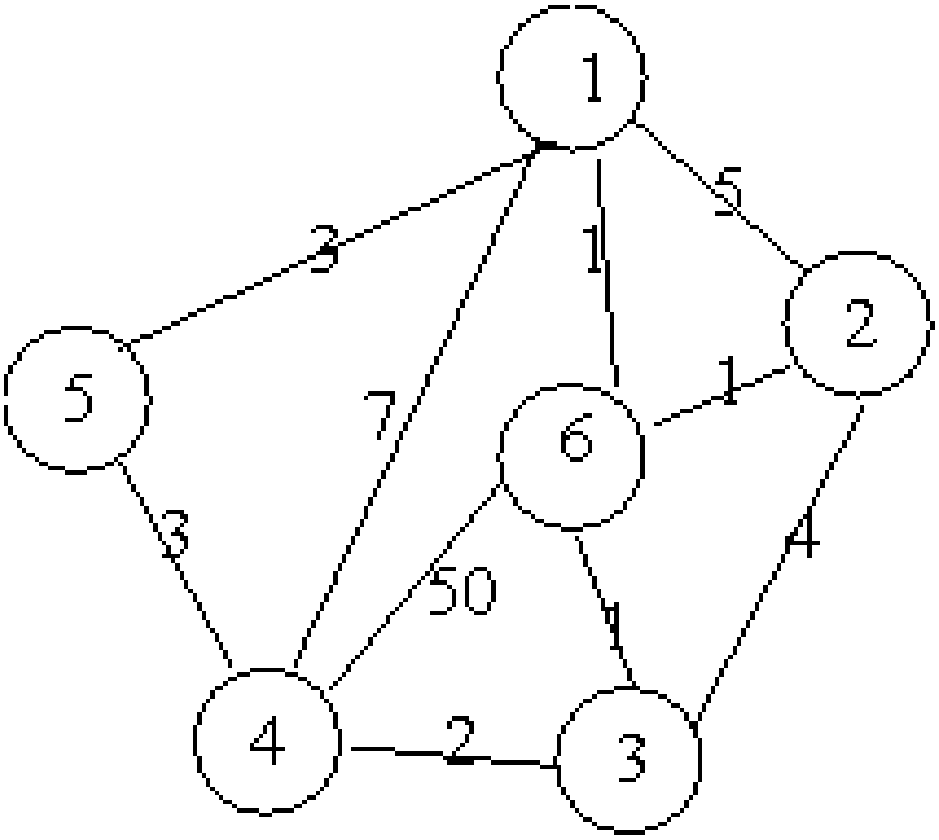
All pairs shortest path

- *The problem:* find the shortest path between every pair of vertices of a graph
- *The graph*: may contain negative edges but no negative cycles

```
1. D = W   // initialize D array to W [ ]
2. P = 0    // initialize P array to [0]
3. for k = 1 to n
4.      do for i = 1 to n
5.          do for j = 1 to n
6.              if (D[ i, j ] > D[ i, k ] + D[ k, j ] )
7.                      then  { D[ i, j ] = D[ i, k ] + D[ k, j ]
8.                              P[ i, j ] = P[k,j];
                              }
```

Complexity $O(|V|^3)$

Floyd Algorithm example



Floyd Algorithm example (cont.)

The final distance matrix and P

$D^6 =$

	1	2	3	4	5	6
1	0	2(6)	2(6)	4(6)	3	1
2	2(6)	0	2(6)	4(6)	5(6)	1
3	2(6)	2(6)	0	2	5(4)	1
4	4(6)	4(6)	2	0	3	3(3)
5	3	5(6)	5(4)	3	0	4(1)
6	1	1	1	3(3)	4(1)	0

The values in parenthesis are the non zero P values.

Summary

- Graph Introduction
- Graph definition
- Graph Terminology
- Graph applications
- Graph Representation
- Graph Traversals
- Shortest Paths
  - Dijsktra algorithm
  - Floyd algorithm

FPT UNIVERSITY
DREAM OF INNOVATION

Reading at home

Text book: Data Structures and Algorithms in Java