CS101 Algorithms and Data Structures

Graphs

Textbook Ch B.4, B.5.1, 22.1

Outline

- Definitions
  - Undirected graphs
  - Directed graph
- Representation
  - Adjacency matrix
  - Adjacency list

Outline

A graph is an abstract data type for storing adjacency relations

- We start with definitions:
  - Vertices, edges, degree and sub-graphs
- We will describe paths in graphs
  - Simple paths and cycles
- Definition of connectedness
- Weighted graphs
- We will then reinterpret these in terms of directed graphs
- Directed acyclic graphs

Undirected Graphs

We will define an Undirected Graph ADT as a collection of *vertices*

$$V = \{v_1, v_2, ..., v_n\}$$

– The number of vertices is denoted by

$$|V| = n$$

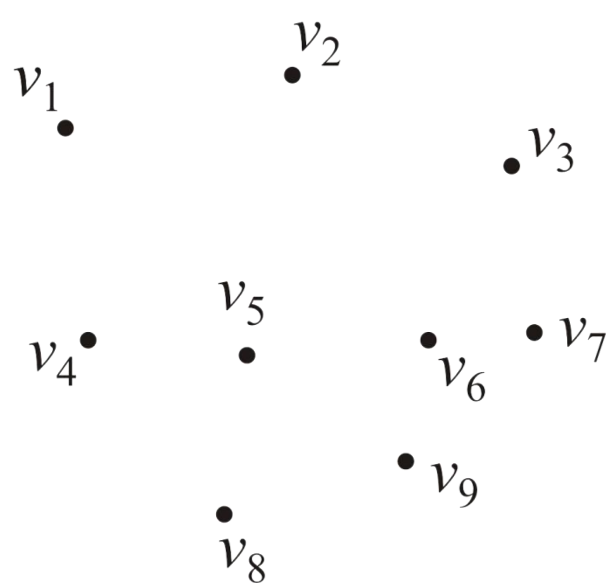
– Associated with this is a collection E of unordered pairs $\{v_i, v_j\}$ termed *edges* which connect the vertices

Undirected Graphs

Consider this collection of vertices

$$V = \{v_1, v_2, \ldots, v_9\}$$

where $|V| = n = 9$

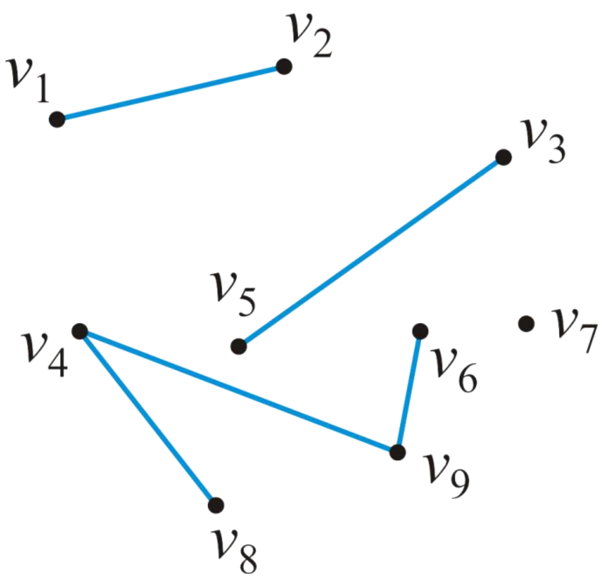


Undirected graphs

Associated with these vertices are $|E| = 5$ edges

$$E = \{\{v_1, v_2\}, \{v_3, v_5\}, \{v_4, v_8\}, \{v_4, v_9\}, \{v_6, v_9\}\}$$

- The pair $\{v_j, v_k\}$ indicates that both vertex v_j is adjacent to vertex v_k and vertex v_k is adjacent to vertex v_j



Undirected graphs

We will assume that a vertex is never adjacent to itself

– For example, $\{v_1, v_1\}$ will not define an edge

The maximum number of edges in an undirected graph is

$$|E| \le \binom{|V|}{2} = \frac{|V|\left(|V|-1\right)}{2} = \mathrm{O}\left(|V|^2\right)$$

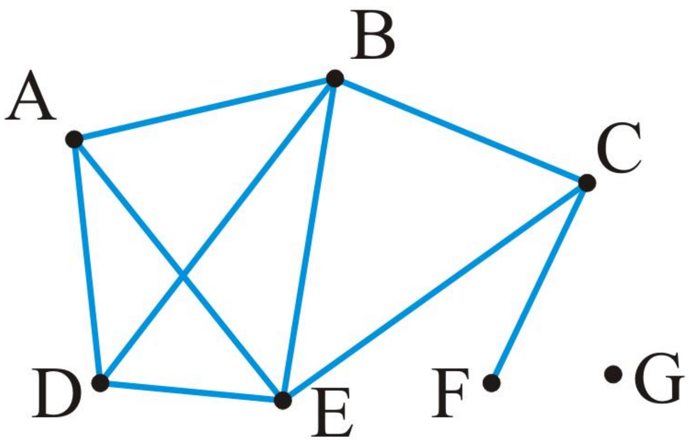
An undirected graph

Example: given the $|V| = 7$ vertices

$$V = \{\mathrm{A, B, C, D, E, F, G}\}$$

and the $|E| = 9$ edges

$$E = \{\{\mathrm{A, B}\}, \{\mathrm{A, D}\}, \{\mathrm{A, E}\}, \{\mathrm{B, C}\}, \{\mathrm{B, D}\}, \{\mathrm{B, E}\}, \{\mathrm{C, E}\}, \{\mathrm{C, F}\}, \{\mathrm{D, E}\}\}$$



Degree

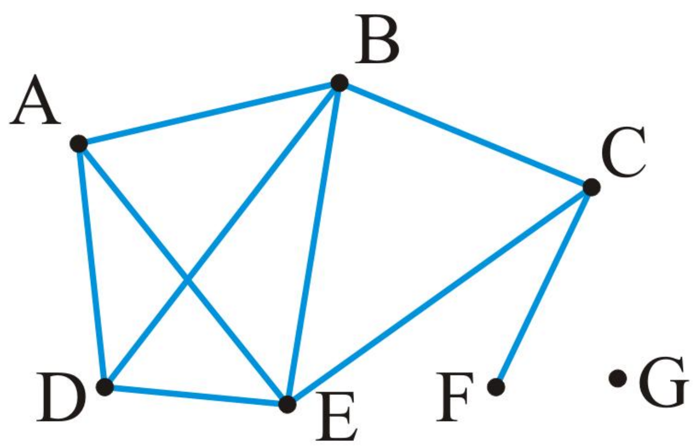
The degree of a vertex is defined as the number of adjacent vertices

degree(A) = degree(D) = degree(C) = 3

degree(B) = degree(E) = 4

degree(F) = 1

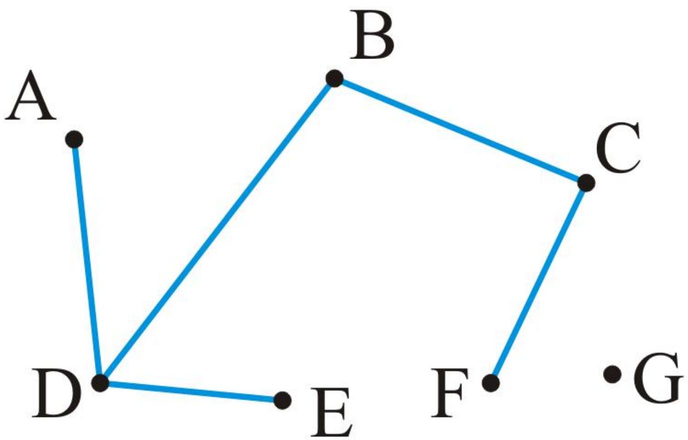
degree(G) = 0

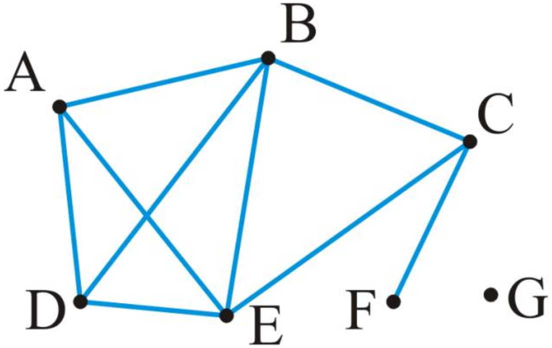


Those vertices adjacent to a given vertex are its *neighbors*

Sub-graphs

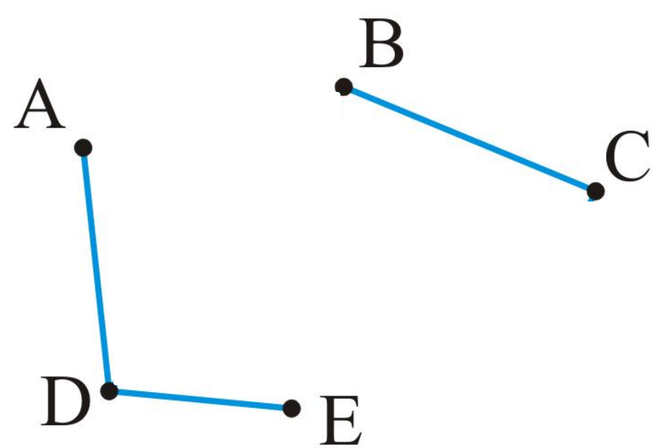
A *sub-graph* of a graph contains a subset of the vertices and a subset of the edges that connect the subset of the vertices in the original graph

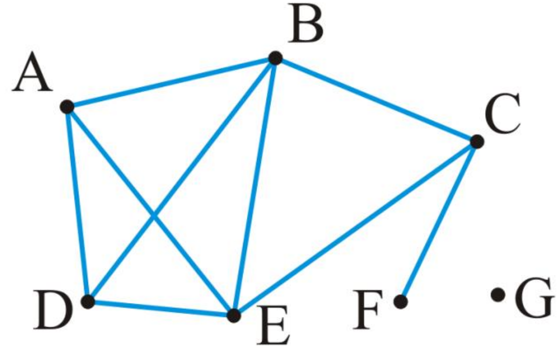




Sub-graphs

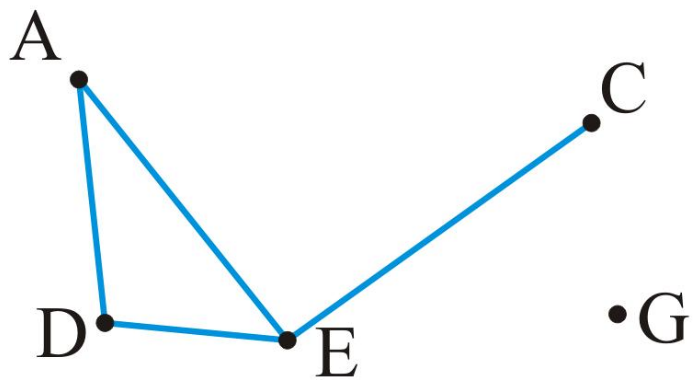
A *sub-graph* of a graph contains a subset of the vertices and a subset of the edges that connect the subset of the vertices in the original graph

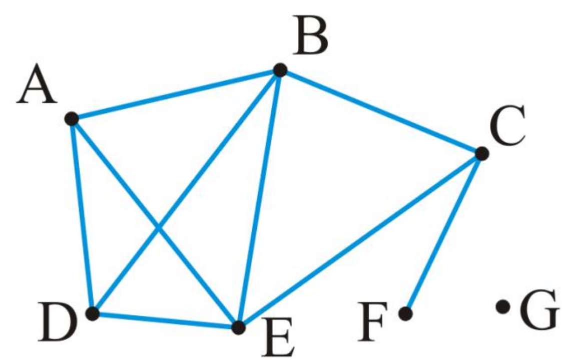




Vertex-induced sub-graphs

A *vertex-induced sub-graph* contains a subset of the vertices and all the edges in the original graph between those vertices





Paths

A path in an undirected graph is an ordered sequence of vertices

$(v_0, v_1, v_2, ..., v_k)$

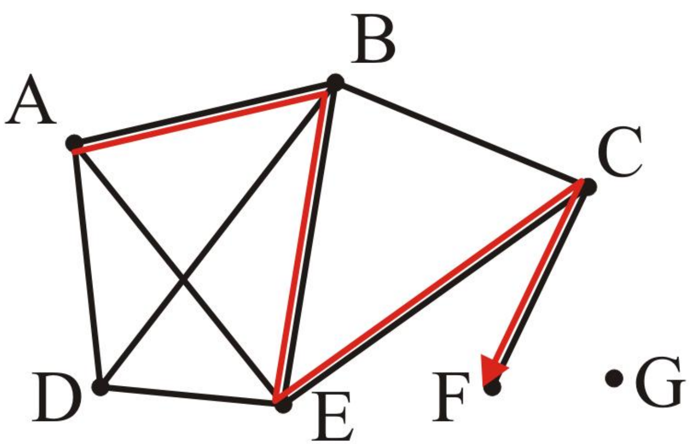
where $\{v_{j-1}, v_j\}$ is an edge for $j = 1, ..., k$

- Termed *a path from* v_0 *to* v_k
- The length of this path is k

Paths

A path of length 4:

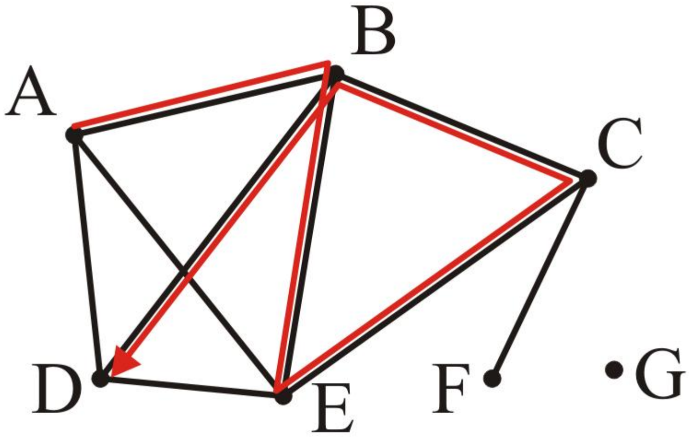
(A, B, E, C, F)



Paths

A path of length 5:

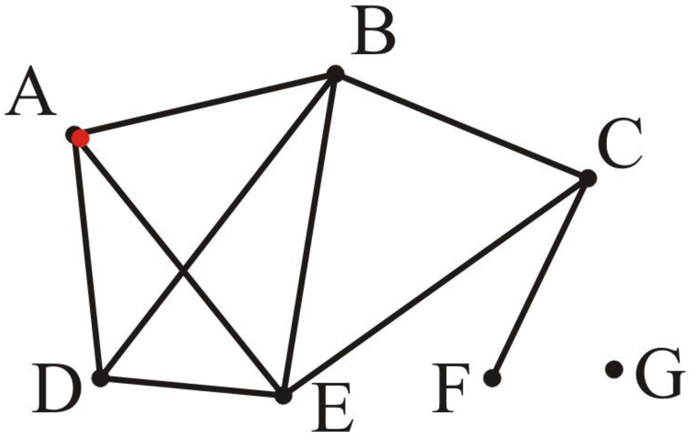
(A, B, E, C, B, D)



Paths

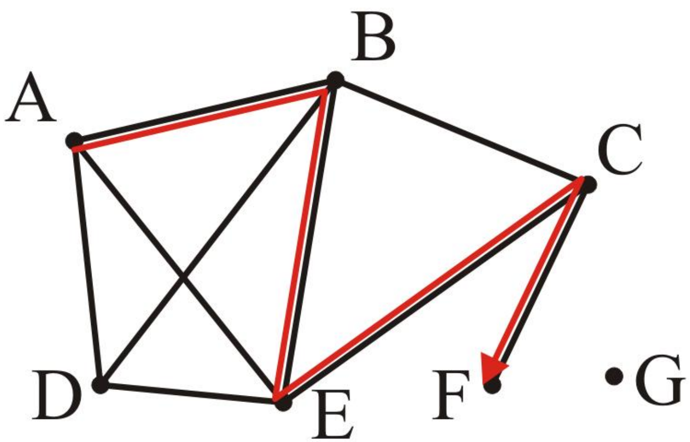
A *trivial* path of length 0:

(A)

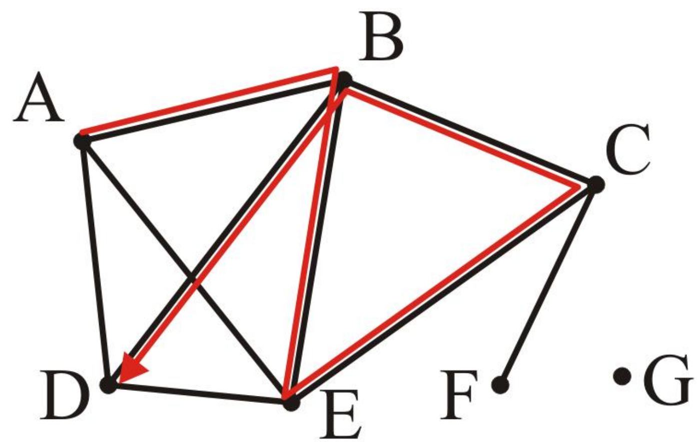


Simple path

A *simple path* has no repetitions (other than perhaps the first and last vertices)



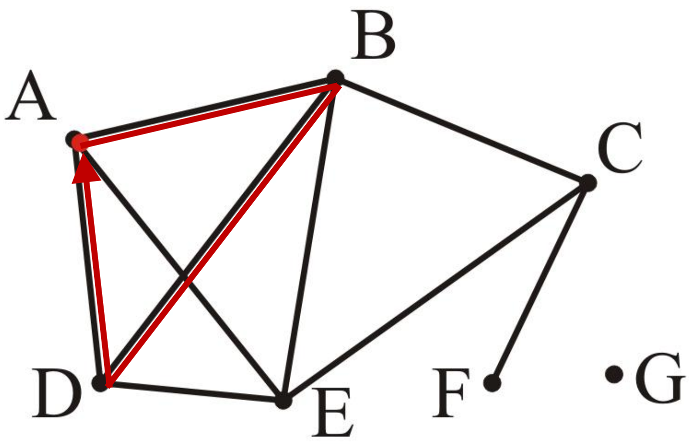
(A, B, E, C, F)



(A, B, E, C, B, D)

Simple cycle

A *simple cycle* is a simple path of at least two vertices with the first and last vertices equal

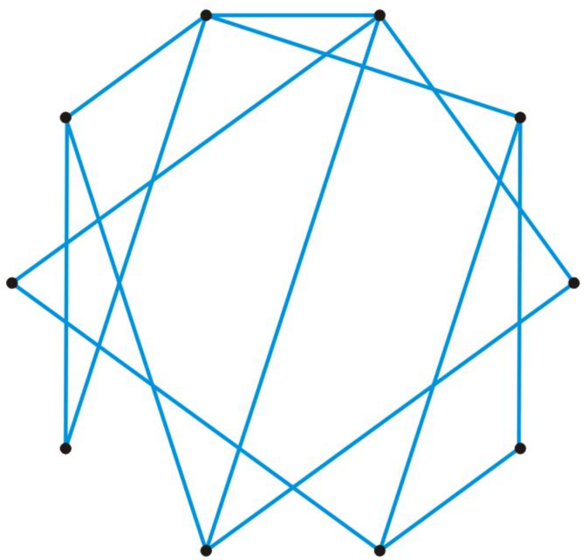


(A, B, D, A)

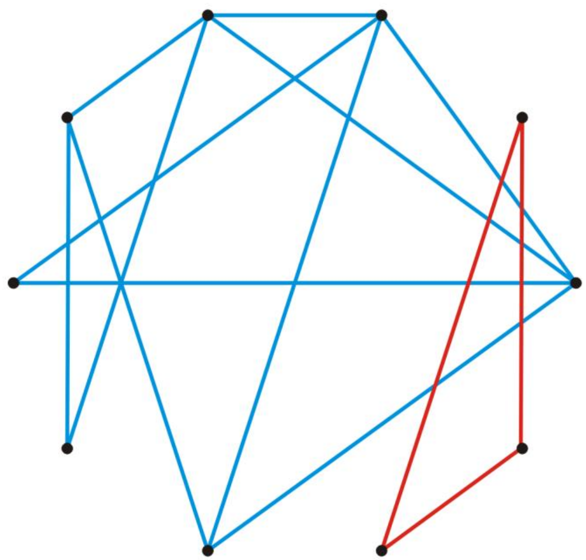
Connectedness

Two vertices v_i, v_j are said to be *connected* if there exists a path from v_i to v_j

A graph is connected if there exists a path between any two vertices



A connected graph

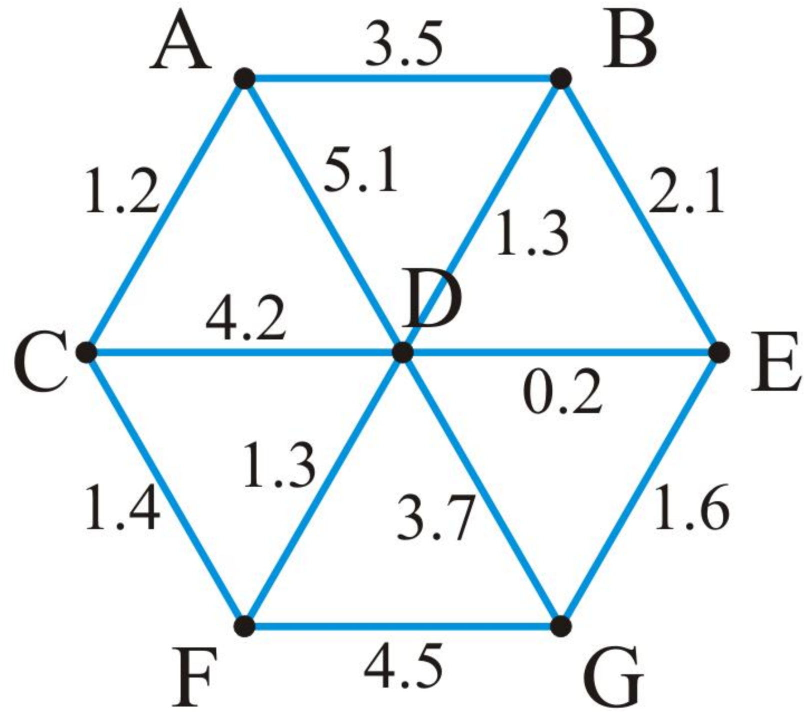


An unconnected graph

Weighted graphs

A weight may be associated with each edge in a graph

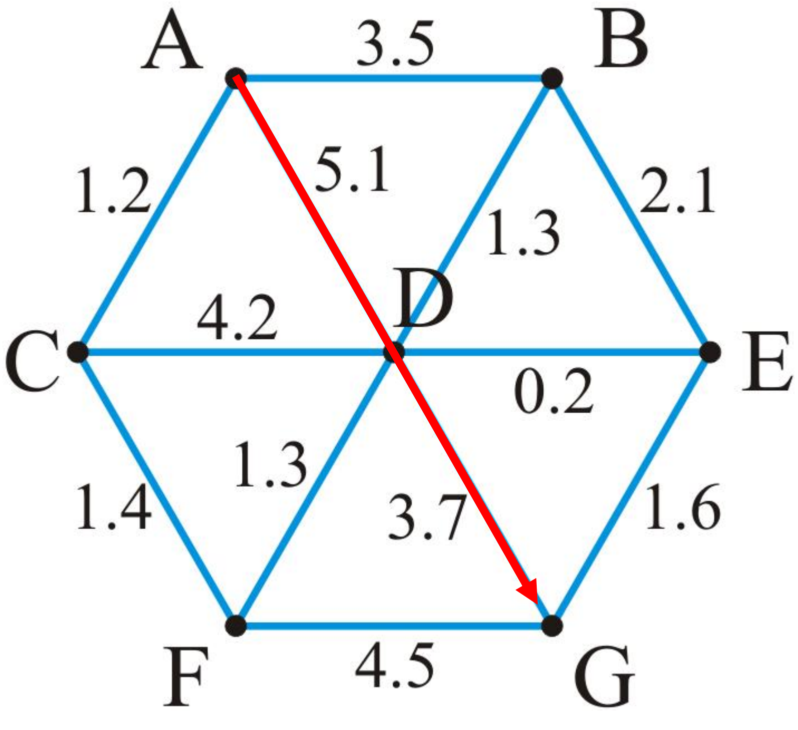
- This could represent distance, energy consumption, cost, etc.
- Such a graph is called a *weighted graph*



Weighted graphs

The *length* of a path within a weighted graph is the sum of all of the edges which make up the path

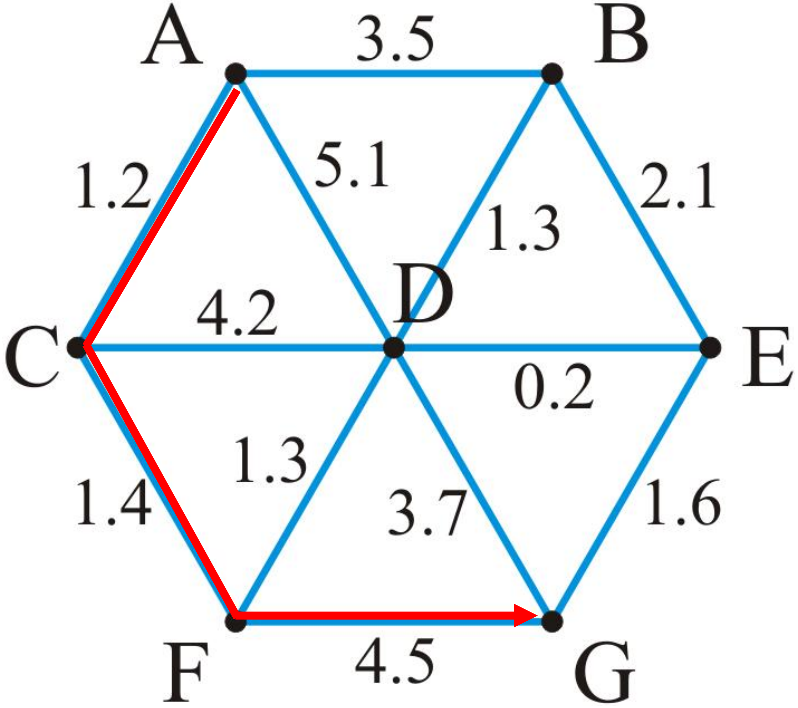
- The length of the path (A, D, G) in the following graph is $5.1 + 3.7 = 8.8$



Weighted graphs

Different paths may have different weights

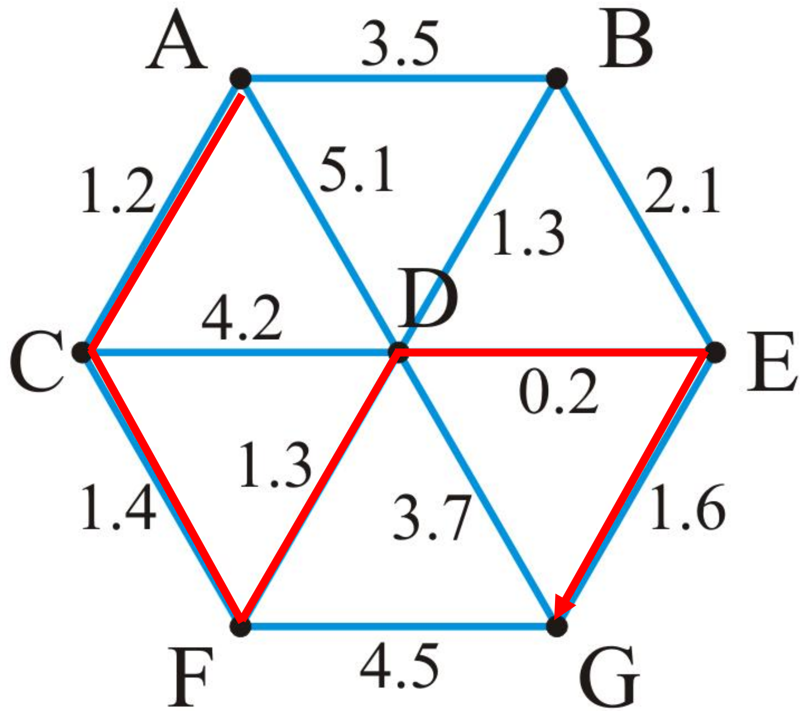
– Another path is (A, C, F, G) with length $1.2 + 1.4 + 4.5 = 7.1$



Weighted graphs

Problem: find the shortest path between two vertices

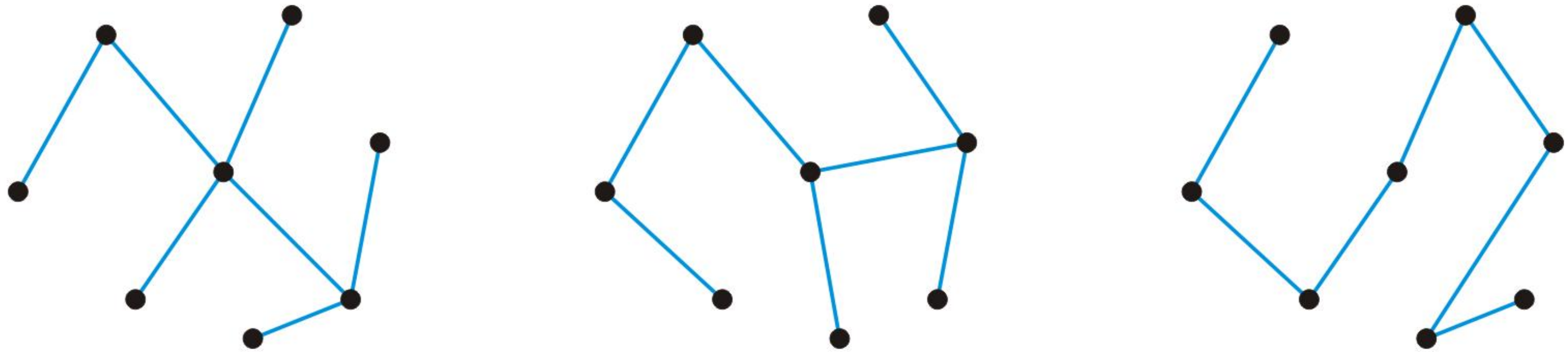
– Here, the shortest path from A to G is (A, C, F, D, E, G) with length 5.7



Trees

A graph is a tree if it is connected and there is a unique path between any two vertices

– Example: three trees on the same eight vertices



Properties:

– The number of edges is $|E| = |V| - 1$

– The graph is *acyclic*, that is, it does not contain any cycles

– Adding one more edge must create a cycle

– Removing any one edge creates two unconnected sub-graphs

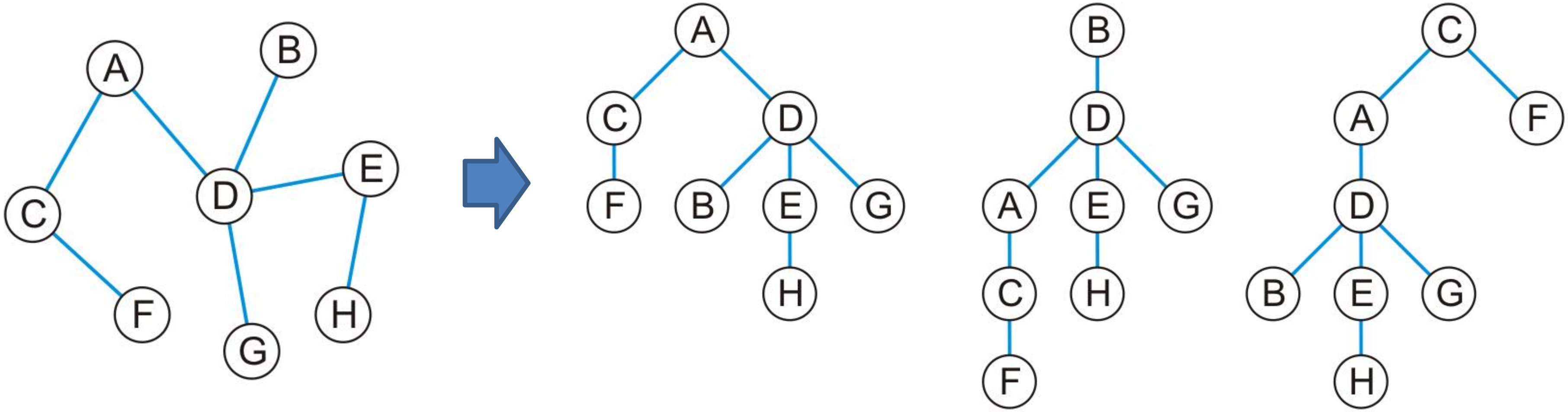
Trees

Any tree can be converted into a rooted tree by:

- Choosing any vertex to be the root
- Defining its neighboring vertices as its children

and then recursively defining:

- All neighboring vertices other than that one designated its parent to be its children



Forests

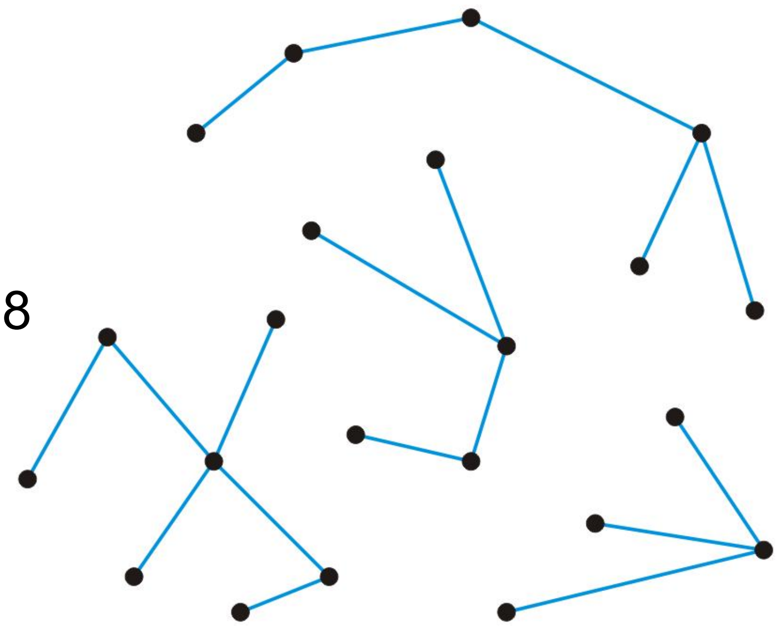
A forest is any graph that has no cycles

Consequences:

- The number of edges is $|E| < |V|$
- The number of trees is $|V| - |E|$
- Removing any one edge adds one more tree to the forest

Here is a forest with 22 vertices and 18 edges

- There are four trees



Outline

- Definitions
  - Undirected graphs
  - Directed graph
- Representation
  - Adjacency matrix
  - Adjacency list

Directed graphs

In a *directed graph*, the edges on a graph are be associated with a direction

- Edges are ordered pairs (v_j, v_k) denoting a connection from v_j to v_k
- The edge (v_j, v_k) is different from the edge (v_k, v_j)

Streets are directed graphs:

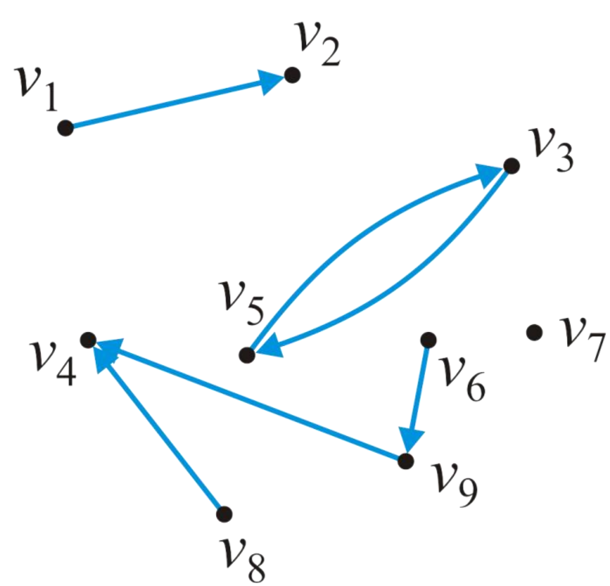
- In most cases, you can go two ways unless it is a one-way street

Directed graphs

Given a graph of nine vertices $V = \{v_1, v_2, \ldots v_9\}$

– These six pairs $(v_j\,,\, v_k)$ are *directed edges*

$$E = \{(v_1, v_2), (v_3, v_5), (v_5, v_3), (v_6, v_9), (v_8, v_4), (v_9, v_4)\}$$



Directed graphs

The maximum number of directed edges in a directed graph is

$$|E| \le 2\binom{|V|}{2} = 2\frac{|V|\left(|V|-1\right)}{2} = |V|\left(|V|-1\right) = \mathrm{O}\left(|V|^2\right)$$

In and out degrees

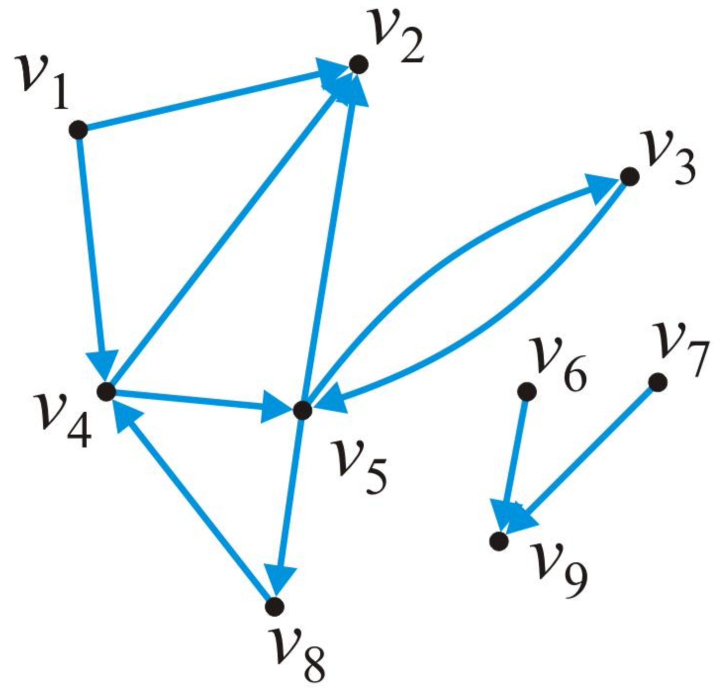
The degree of a vertex in a directed graph:

- The *out-degree* of a vertex is the number of outward edges from the vertex
- The *in-degree* of a vertex is the number of inward edges to the vertex

In this graph:

in_degree(v_1) = 0   out_degree(v_1) = 2

in_degree(v_5) = 2   out_degree(v_5) = 3



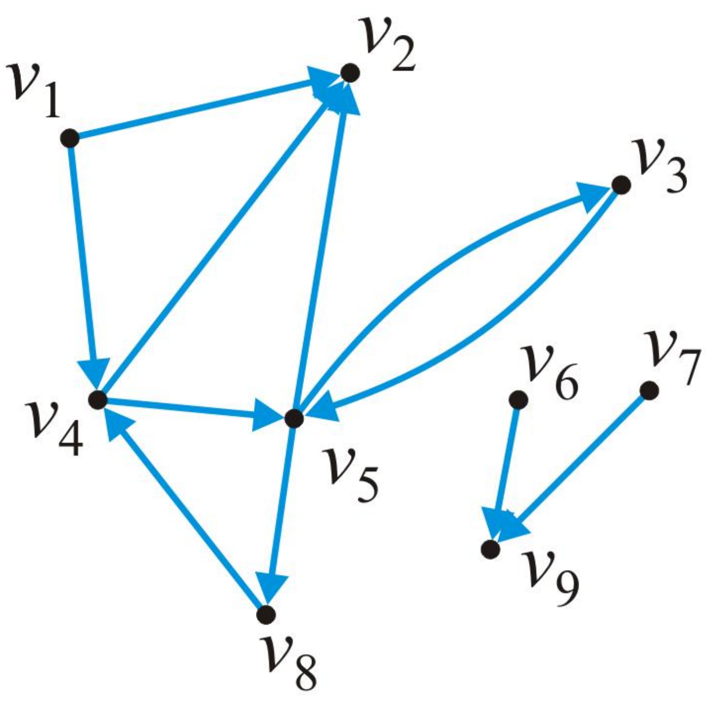
Sources and sinks

Definitions:

- Vertices with an in-degree of zero are described as *sources*
- Vertices with an out-degree of zero are described as *sinks*

In this graph:

- Sources: v_1, v_6, v_7
- Sinks: v_2, v_9



Paths

A path in a directed graph is an ordered sequence of vertices

$(v_0, v_1, v_2, ..., v_k)$

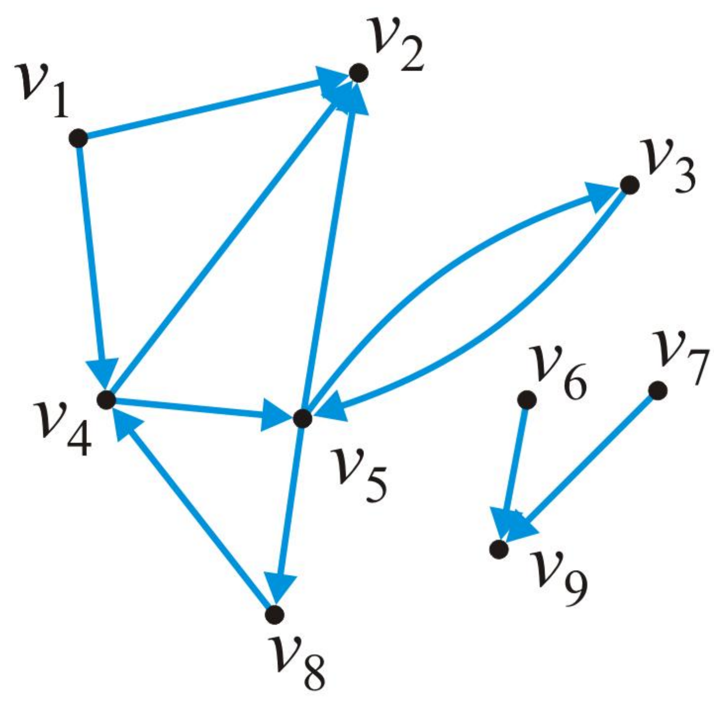
where (v_{j-1}, v_j) is an edge for $j = 1, ..., k$

A path of length 5 in this graph is

$(v_1, v_4, v_5, v_3, v_5, v_2)$

A simple cycle of length 3 is

(v_8, v_4, v_5, v_8)



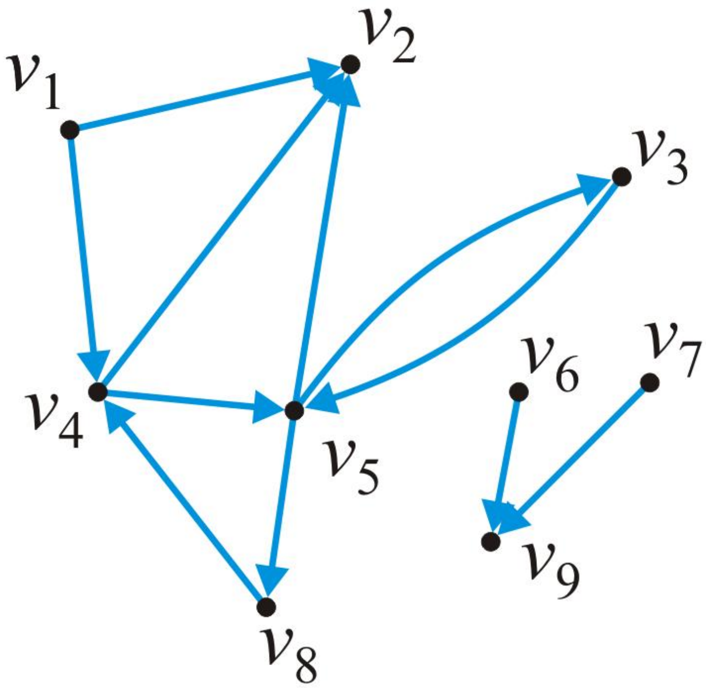
Connectedness

Two vertices v_j, v_k are said to be *connected* if there exists a path from v_j to v_k

- A graph is *strongly connected* if there exists a directed path between any two vertices
- A graph is *weakly connected* there exists a path between any two vertices that ignores the direction

In this graph:

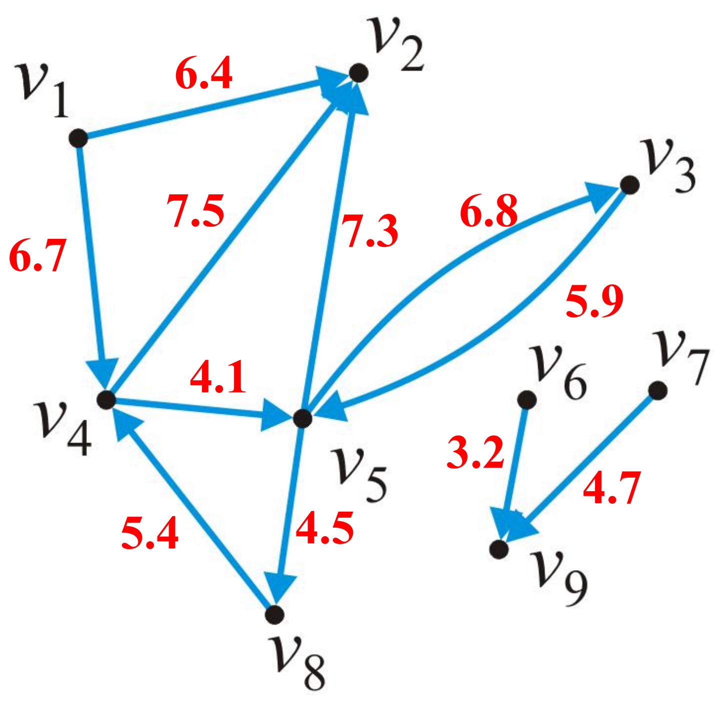
- The sub-graph $\{v_3, v_4, v_5, v_8\}$ is strongly connected
- The sub-graph $\{v_1, v_2, v_3, v_4, v_5, v_8\}$ is weakly connected



Weighted directed graphs

In a weighted directed graphs, each edge is associated with a value

If both (v_j, v_k) and (v_k, v_j) are edges, it is not required that they have the same weight

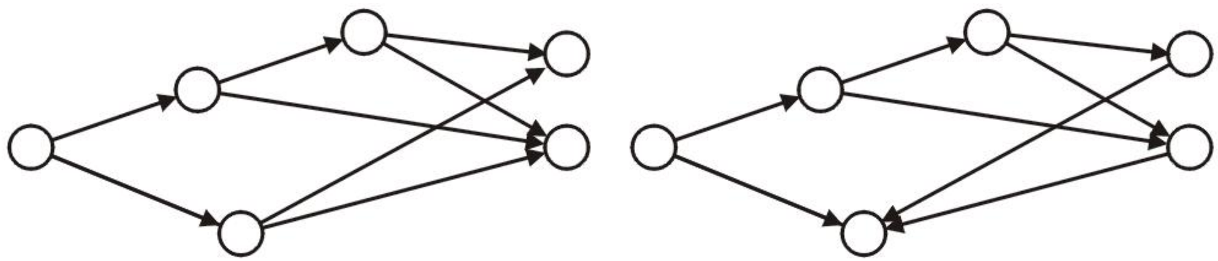


Directed acyclic graphs

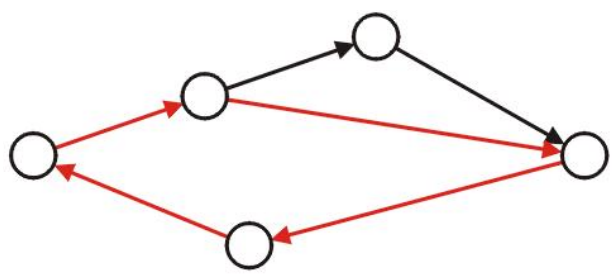
A *directed acyclic graph* is a directed graph which has no cycle

- These are commonly referred to as DAGs
- They are graphical representations of partial orders on a finite number of elements

These two are DAGs:



This directed graph is not acyclic:



Directed acyclic graphs

Applications of directed acyclic graphs include:

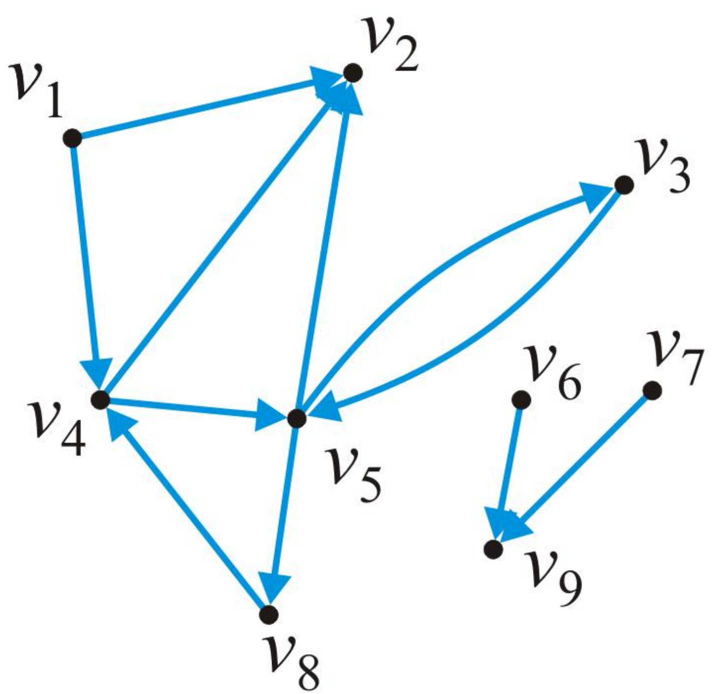
- The parse tree constructed by a compiler
- A reference graph that can be garbage collected using simple reference counting
- Dependency graphs such as those used in instruction scheduling and **`makefiles`**
- Dependency graphs between classes formed by inheritance relationships in object-oriented programming languages
- Information categorization systems, such as folders in a computer
- Directed acyclic word graph data structure to memory-efficiently store a set of strings (words)

Reference: http://en.wikipedia.org/wiki/Directed_acyclic_graph

Representations

How do we store the adjacency relations?

- Binary-relation list
- Adjacency matrix
- Adjacency list



Summary

In this topic, we have covered:

- Basic graph definitions
  - Vertex, edge, degree, adjacency
- Paths, simple paths, and cycles
- Connectedness
- Weighted graphs
- Directed graphs
- Directed acyclic graphs

We will continue by looking at a number of problems related to graphs

References

Wikipedia, http://en.wikipedia.org/wiki/Adjacency_matrix
http://en.wikipedia.org/wiki/Adjacency_list

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd Ed., Addison Wesley, 1997, §2.2.1, p.238.
[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, §11.1, p.200.
[3] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, §3.6, p.94.
[4] David H. Laidlaw, Course Notes, http://cs.brown.edu/courses/cs016/lectures/13%20Graphs.pdf

Outline

- Definitions
  - Undirected graphs
  - Directed graph
- Representation
  - Adjacency matrix
  - Adjacency list

The Graph ADT

The Graph ADT describes a container storing an adjacency relation

- Queries include:
  - The number of vertices
  - The number of edges
  - List the vertices adjacent to a given vertex
  - Are two vertices adjacent?
  - Are two vertices connected?

- Modifications include:
  - Inserting or removing an edge
  - Inserting or removing a vertex (and all edges containing that vertex)

The run-time of these operations will depend on the representation

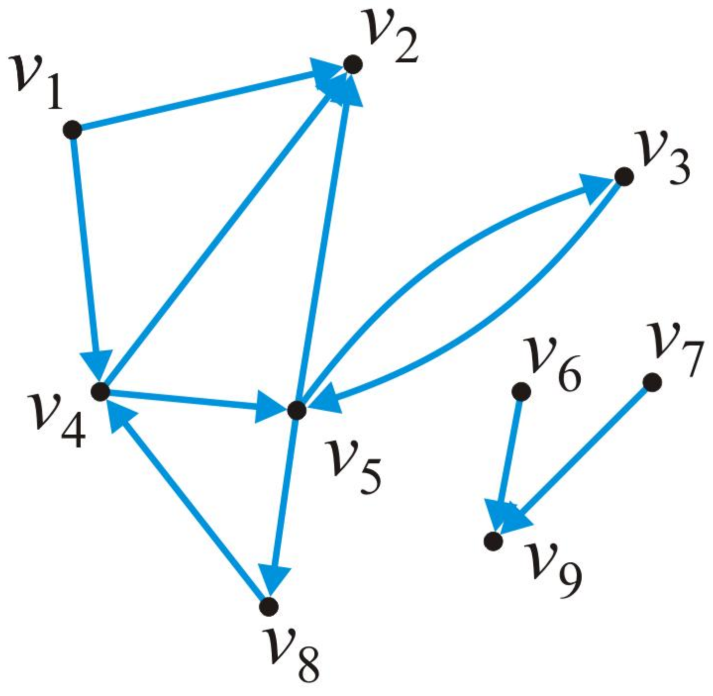
Binary-relation list

The most inefficient is a relation list:

- A container storing the edges

  $\{(1, 2), (1, 4), (3, 5), (4, 2), (4, 5), (5, 2), (5, 3), (5, 8), (6, 9), (7, 9), (8, 4)\}$

- Requires $\Theta(|E|)$ memory
- Determining if v_j is adjacent to v_k is $\mathrm{O}(|E|)$
- Finding all neighbors of v_j is $\Theta(|E|)$



Adjacency matrix

Requiring more memory but also faster, an adjacency matrix

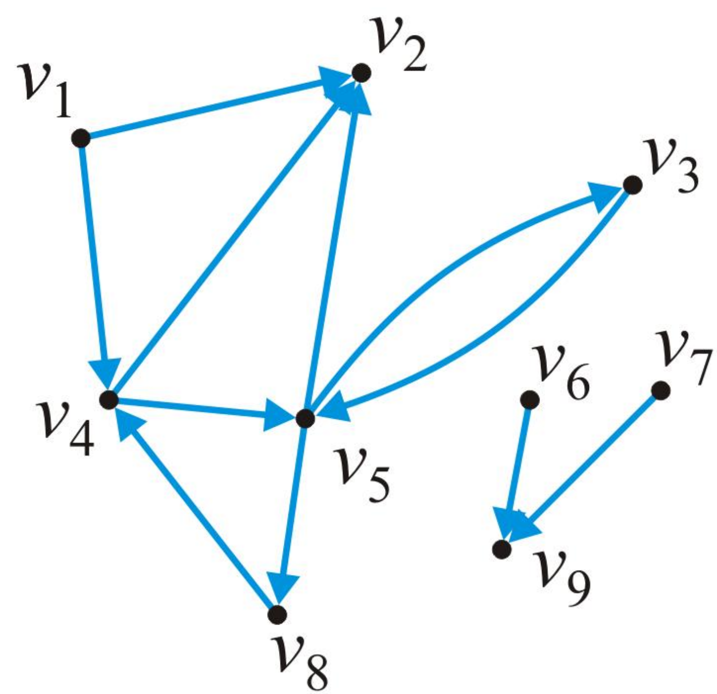
– The matrix entry (j, k) is set to true if there is an edge (v_j, v_k)

	1	2	3	4	5	6	7	8	9
1		T		T					
2									
3					T				
4		T			T				
5		T	T					T	
6									T
7									T
8				T					
9									

– Requires $\Theta(|V|^2)$ memory

– Determining if v_j is adjacent to v_k is $O(1)$

– Finding all neighbors of v_j is $\Theta(|V|)$

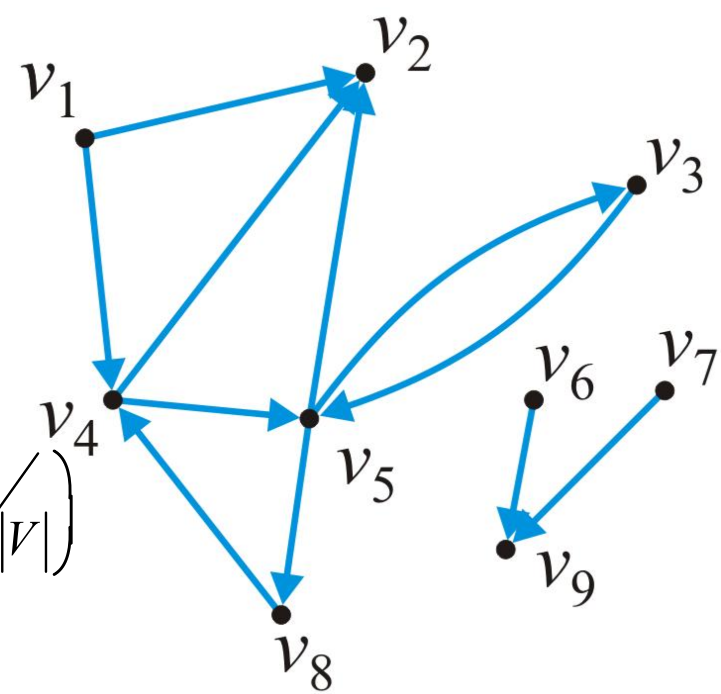


Adjacency list

Most efficient for algorithms is an adjacency list

– Each vertex is associated with a list of its neighbors

$$
\begin{array}{ll}
1 & \bullet \to 2 \to 4 \\
2 & \bullet \\
3 & \bullet \to 5 \\
4 & \bullet \to 2 \to 5 \\
5 & \bullet \to 2 \to 3 \to 8 \\
6 & \bullet \to 9 \\
7 & \bullet \to 9 \\
8 & \bullet \to 4 \\
9 & \bullet
\end{array}
$$



– Requires $\Theta(|V| + |E|)$ memory

– On average:

- Determining if v_j is adjacent to v_k is $\mathrm{O}\left(\frac{|E|}{|V|}\right)$
- Finding all neighbors of v_j is $\Theta\left(\frac{|E|}{|V|}\right)$

Outline

- In this topic, we will cover the representation of graphs on a computer
- We will examine:
  - an adjacency matrix representation
  - smaller representations and pointer arithmetic
  - sparse matrices and linked lists

Adjacency Matrix

A graph of n vertices may have up to

$$\binom{n}{2} = \frac{n(n-1)}{2} = \mathbf{O}(n^2)$$

edges

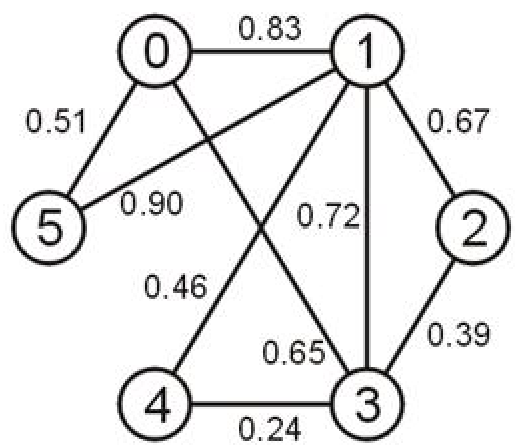
The first straight-forward implementation is an adjacency matrix

Adjacency Matrix

Define an $n \times n$ matrix $\mathbf{A} = (a_{ij})$ and if the vertices v_i and v_j are connected with weight w, then set $a_{ij} = w$ and $a_{ji} = w$

That is, the matrix is symmetric, *e.g.*,

	0	1	2	3	4	5
0		0.83		0.65		0.51
1	0.83		0.67	0.72	0.46	0.90
2		0.67		0.39		
3	0.65	0.72	0.39		0.24	
4		0.46		0.24		
5	0.51	0.90				



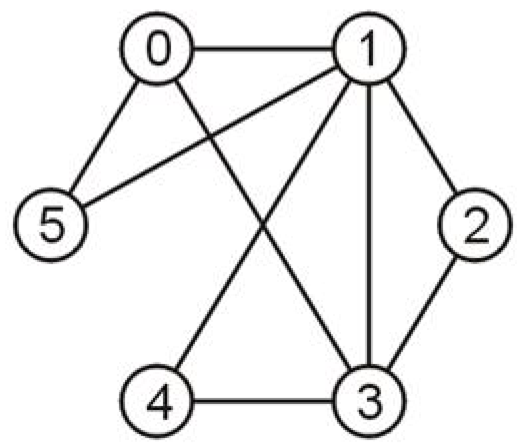
Adjacency Matrix

An unweighted graph may be saved as an array of Boolean values

– vertices v_i and v_j are connected then set

$$a_{ij} = a_{ji} = true$$

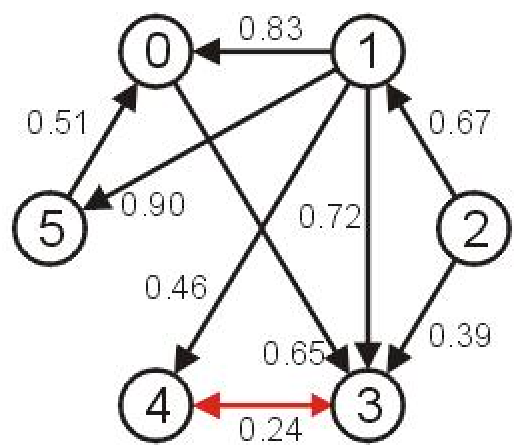
	0	1	2	3	4	5
0		T	F	T	F	T
1	T		T	T	T	T
2	F	T		T	F	F
3	T	T	T		T	F
4	F	T	F	T		F
5	T	T	F	F	F	



Adjacency Matrix

If the graph was directed, then the matrix would not necessarily be symmetric

	0	1	2	3	4	5
0				0.65		
1	0.83			0.72	0.46	0.90
2		0.67		0.39		
3					0.24	
4				0.24		
5	0.51					



Adjacency Matrix

First we must allocate memory for a two-dimensional array

C++ does not have native support for anything more than one-dimensional arrays, thus how do we store a two-dimensional array?

- as an array of arrays

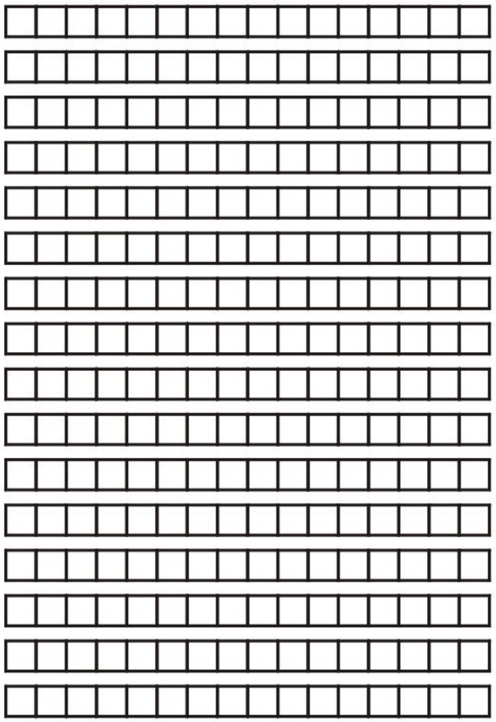
Adjacency Matrix

Suppose we require a 16 × 16 matrix of double-precision floating-point numbers

Each row of the matrix can be represented by an array

The address of the first entry must be stored in a pointer to a double:

```
double *
```



Adjacency Matrix

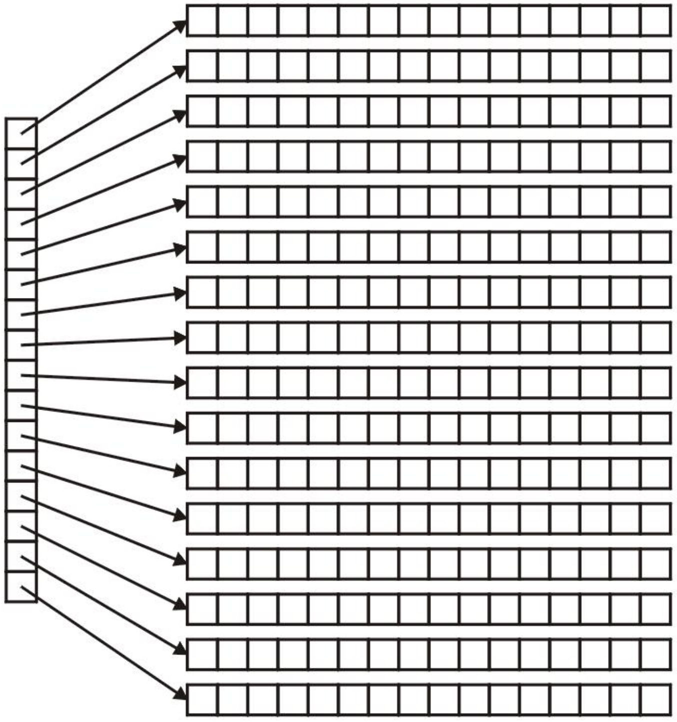
However, because we must store 16 of these pointers-to-doubles, it makes sense that we store these in an array

What is the declaration of this array?

Well, we must store a

pointer to a pointer to a double

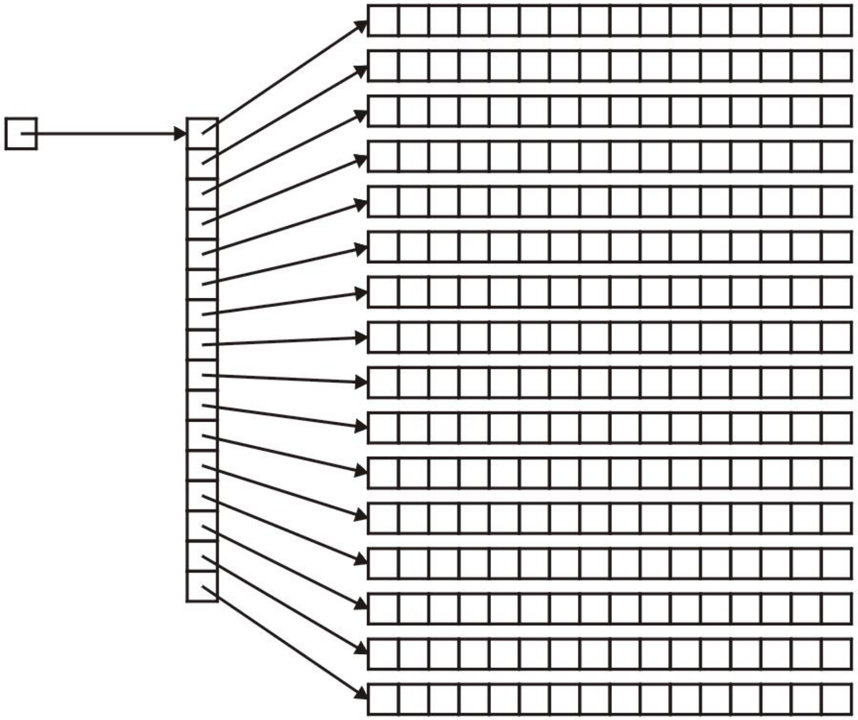
That is: double **



Adjacency Matrix

Thus, the address of the first array must be declared to be:

`double **matrix;`

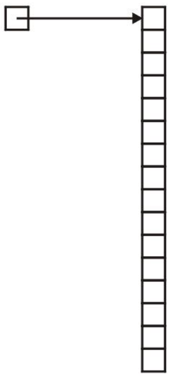


Adjacency Matrix

The next question is memory allocation

First, we must allocate the memory for the array of pointers to doubles:

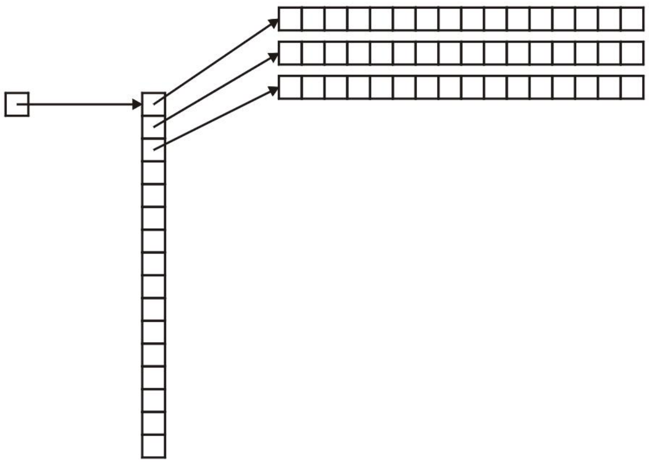
```
matrix = new double * [16];
```



Adjacency Matrix

Next, to each entry of this matrix, we must assign the memory allocated for an array of doubles

```
for ( int i = 0; i < 16; ++i ) {
    matrix[i] = new double[16];
}
```



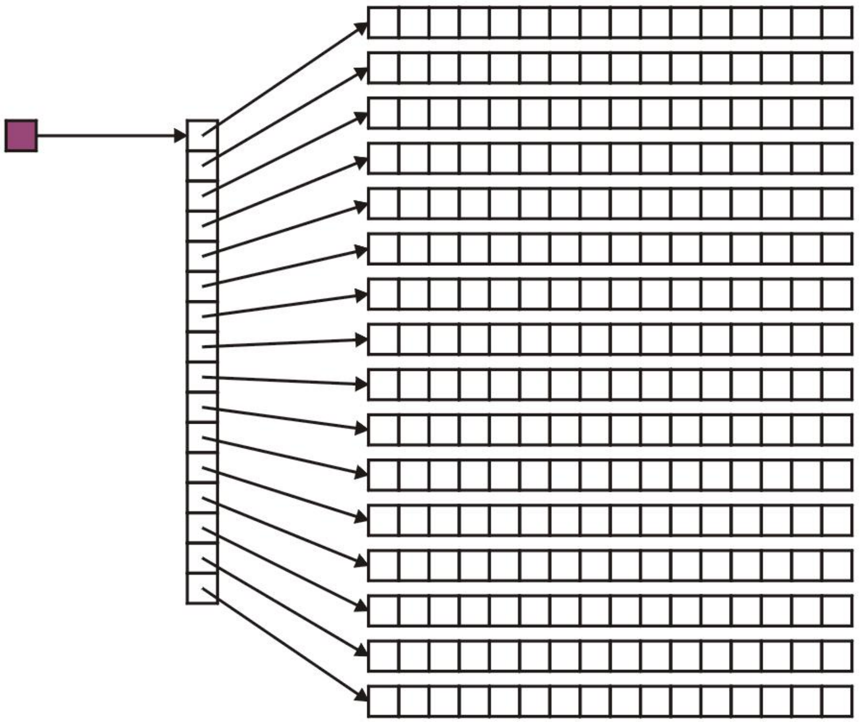
Adjacency Matrix

Accessing a matrix is done through a double index, *e.g.*, `matrix[3][4]`

You can interpret this as `(matrix[3])[4]`

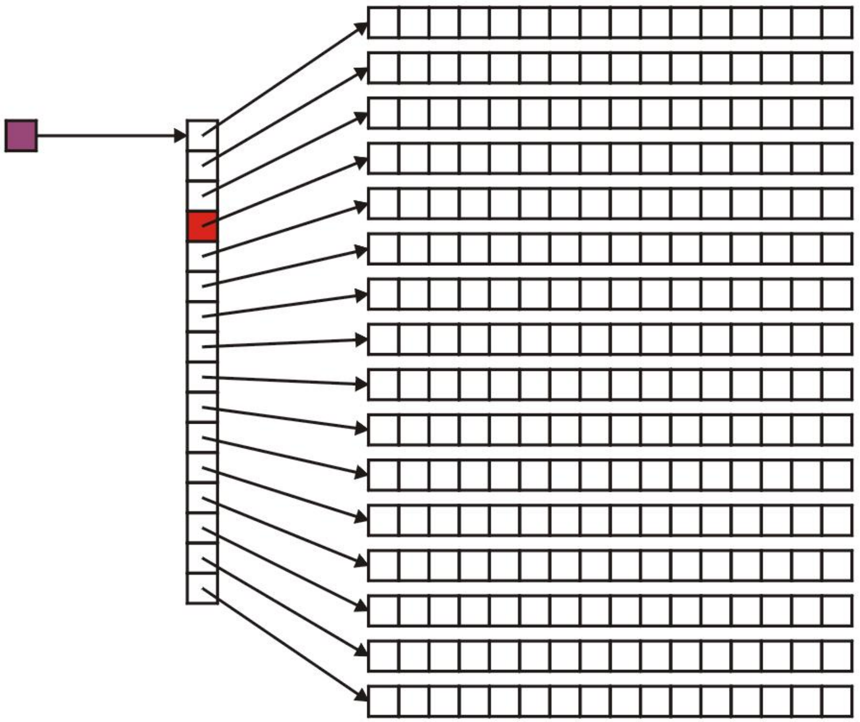
Adjacency Matrix

Recall that in **matrix[3][4]**, the variable matrix is a pointer-to-a-pointer-to-a-double:



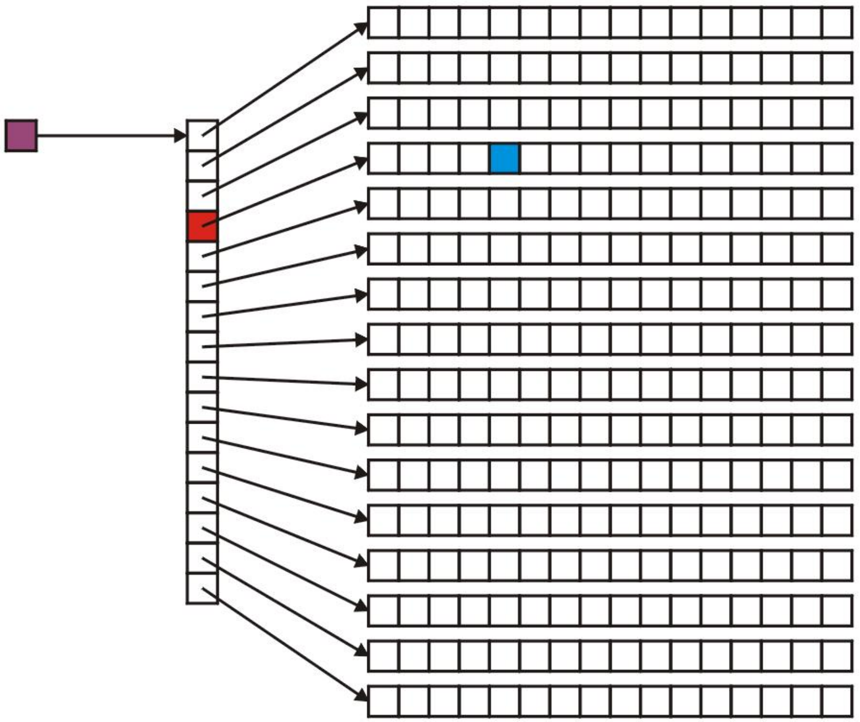
Adjacency Matrix

Therefore, matrix[3] is a pointer-to-a-double:



Adjacency Matrix

And consequently, matrix[3][4] is a double:



C++ Notation Warning

Do not use **matrix[3, 4]** because:

- in C++, the comma operator evaluates the operands in order from left-to-right
- the *value* is the last one

Therefore, **matrix[3, 4]** is equivalent to calling **matrix[4]**

Try it:

```
int i = (3, 4);
cout << i << endl;
```

C++ Notation Warning

Many things will compile if you try to use this notation:

```
matrix = new double[N, N];
```

will allocate an array of N doubles, just like:

```
matrix = new double[N];
```

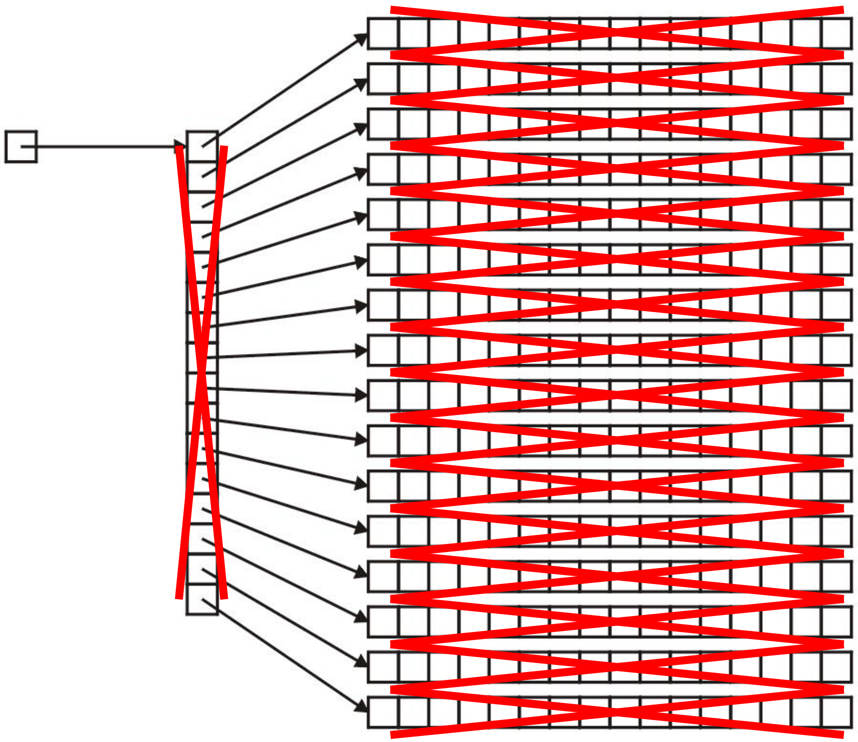
However, this is likely not to do what you really expect...

Adjacency Matrix

Recall that for each call to new[],you must have a corresponding call to delete[]

Therefore, we must use a for-loop to delete the arrays

– implementation up to you



Default Values

Question: what do we do about vertices which are not connected?

- the value 0
- a negative number, *e.g.*, -1
- positive infinity: ∞

The last is the most logical, in that it makes sense that two vertices which are not connected have an infinite distance between them

The distance from a node to itself is 0

Default Values

To use infinity, you may declare a constant static member variable INF:

```
#include <limits>

class Weighted_graph {
    private:
        static const double INF;
        // ...
    // ...
};

const double Weighted_graph::INF =
    std::numeric_limits<double>::infinity();
```

Default Values

As defined in the IEEE 754 standard, the representation of the double-precision floating-point infinity eight bytes:

`0x 7F F0 00 00 00 00 00 00`

Incidentally, negative infinity is stored as:

`0x FF F0 00 00 00 00 00 00`

Default Values

In this case, you can initialize your array as follows:

```
for ( int i = 0; i < N; ++i ) {
    for ( int j = 0; j < N; ++j ) {
        matrix[i][j] = INF;
    }

    matrix[i][i] = 0;
}
```

It makes intuitive sense that the distance from a node to itself is 0

Default Values

If we are representing an unweighted graph, use Boolean values:

```
for ( int i = 0; i < N; ++i ) {
    for ( int j = 0; j < N; ++j ) {
        matrix[i][j] = false;
    }

    matrix[i][i] = true;
}
```

It makes intuitive sense that a vertex is connected to itself

Sparse Matrices

- The memory required for creating an $n \times n$ matrix using a 2D array is $\Theta(n^2)$ bytes
- This could potentially waste a significant amount of memory:
  - Consider a friendship graph: nodes represent persons and edges represent friendship
  - The world population is 7.4 billion => the size of the matrix is $(7.4\times10^9)^2 \approx 55\times10^{18}$
  - However, each person on average has, say, 100 friends. Hence only $\frac{100}{7.4\times10^9}$ of the matrix elements are true. The other elements are the default value: false.

Sparse Matrices

- Matrices where less than 5% of the entries are not the default value (either infinity or 0, or perhaps some other default value) are said to be *sparse*
- Matrices where most entries (25% or more) are not the default value are said to be *dense*
- Clearly, these are not hard limits

Adjacency list

- For an undirected graph, use an array of linked lists to store edges
  - Each vertex has a linked list that stores all the edges connected to the vertex
  - Each node in a linked list must store two items of information: the connecting vertex and the weight

Adjacency list

We may create a new class which stores a vertex-edge pair

```
class Pair {
    private:
        double edge_weight;
        int adacent_vertex;
    public:
        Pair( int, double );
        double weight() const;
        int vertex() const;
};
```

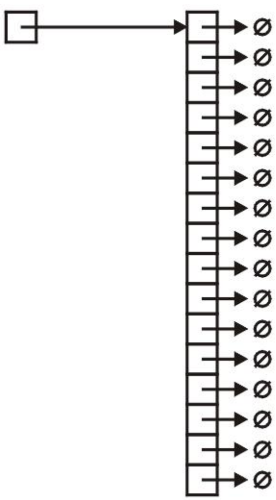
Now create an array of linked-lists storing these pairs

Adjacency list

Thus, we define and create the array:

```
SingleList<Pair> * array;

array = new SingleList<Pair>[16];
```



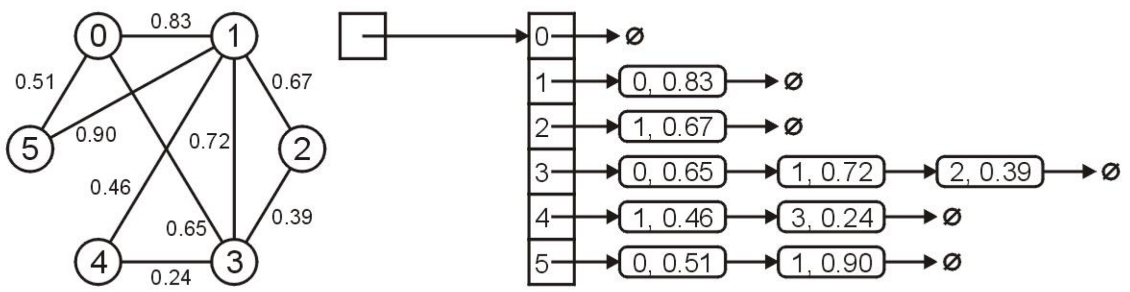
Adjacency list

To reduce redundancy, we would only insert the pair into the linked list corresponding to the larger vertex

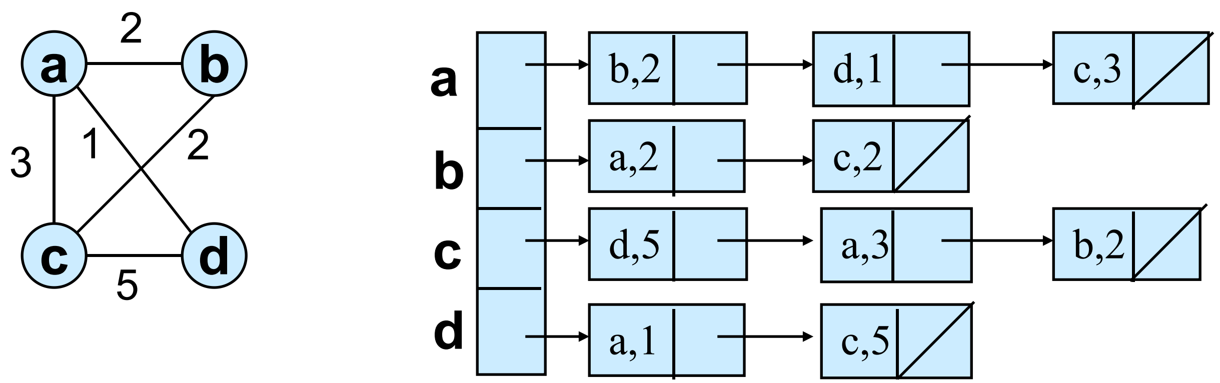
```
void insert( int i, int j, double w ) {
    if ( i < j ) {
        array[j].push_front( Pair(i, w) );
    } else {
        array[i].push_front( Pair(j, w) );
    }
}
```

Adjacency list

For example, the graph shown below would be stored as



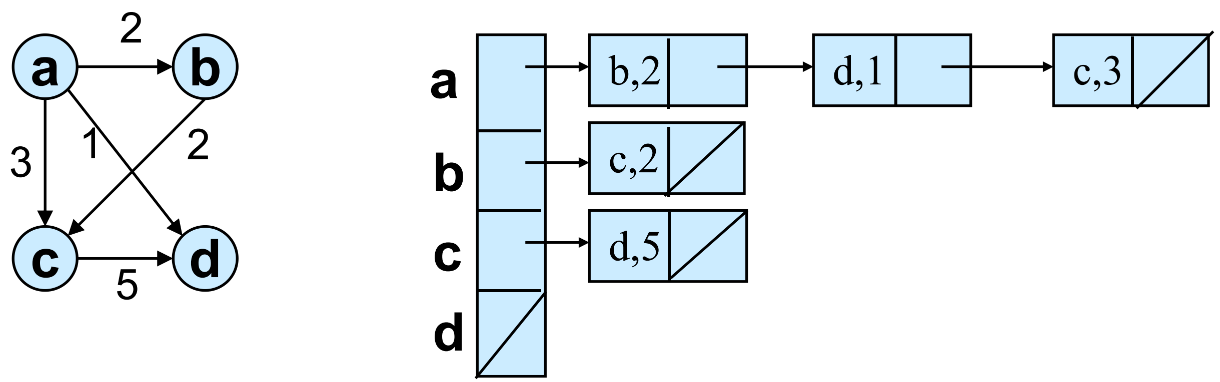
Adjacency list



Adjacency list

- To store a directed graph
  - Each vertex has a linked list that stores all the edges originated from the vertex
  - Each node in a linked list stores two items of information: the vertex that the edge connects to, the weight

Adjacency list



Summary

- In this laboratory, we have looked at a number of graph representations
- C++ lacks a *matrix* data structure
  - must use array of arrays
- The possible factors affecting your choice of data structure are:
  - weighted or unweighted graphs
  - directed or undirected graphs
  - dense or sparse graphs

References

Wikipedia, http://en.wikipedia.org/wiki/Adjacency_matrix
http://en.wikipedia.org/wiki/Adjacency_list

Summary

- Definitions
  - Undirected graphs
  - Directed graph
- Representation
  - Adjacency matrix
  - Adjacency list