Data Structures

Graphs - Introduction

Today’s Lecture

- Introduction to graph

Graph definitions

- A graph is a collection of nodes (or vertices, singular is vertex) and edges (or arcs)
  - Each **node/vertex** contains an element
  - **Edge/arc** :A pair of vertices representing a connection between two nodes in a graph

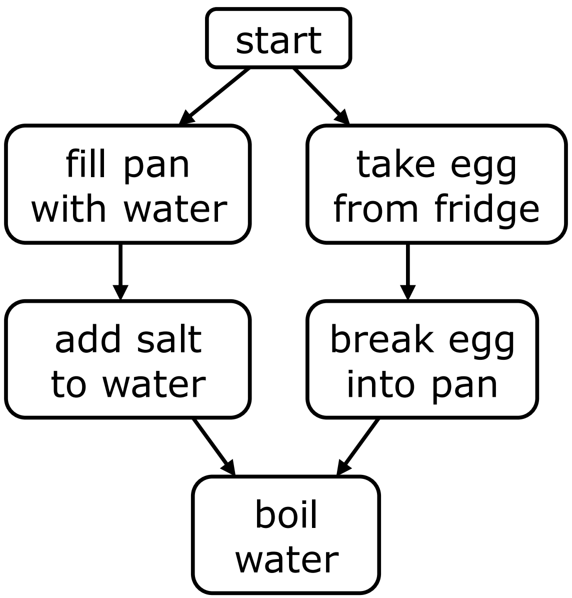
Graphs

- The Graph Data Structure
  - set V of vertices
  - collection E of edges
    (pairs of vertices in V)
- Drawing of a Graph
  - Vertex →circle/oval/node
  - Edge, → line connecting the vertex pair
- V(G) is a finite, nonempty set of vertices
- E(G) is a set of edges (written as pairs of vertices)

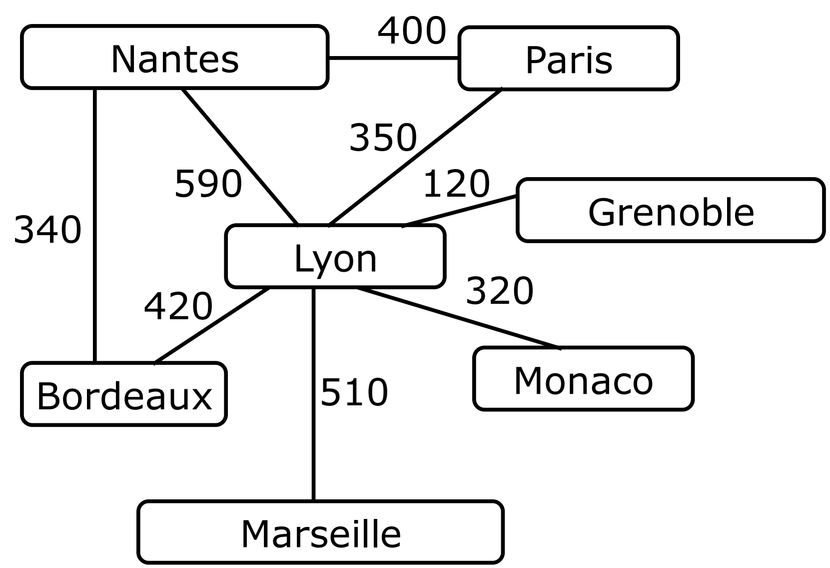
Directed And Undirected Graphs

- There are two kinds of graphs: directed graphs (sometimes called digraphs) and undirected graphs
  - A directed graph/ digraph is one in which the edges have a direction
  - An undirected graph is one in which the edges do not have a direction.

Directed And Undirected Graphs



A directed graph

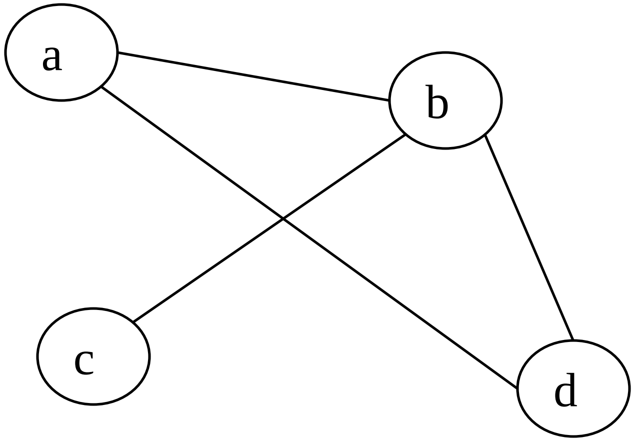


An undirected graph

Undirected Graph Example

Graph G=(V,E):

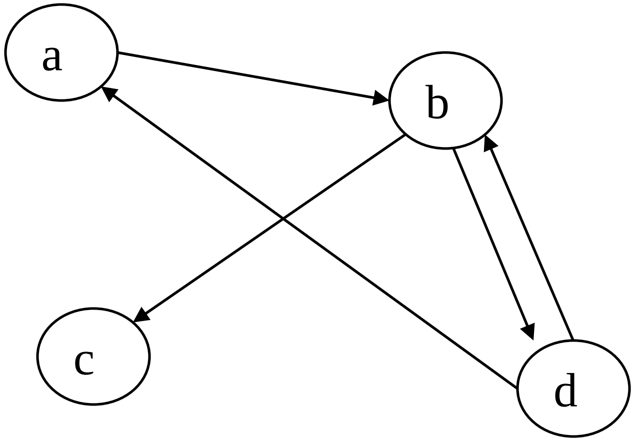
V={a,b,c,d}, E={(a,b),(b,c),(b,d),(a,d)}



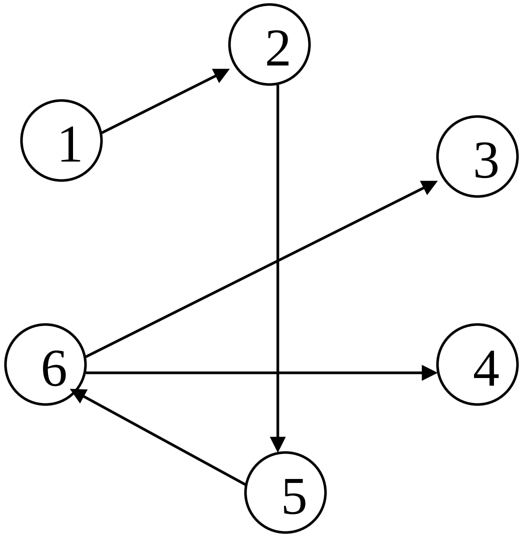
Directed Graph

Graph G=(V,E):

V={a,b,c,d}, E={(a,b),(b,c),(b,d),(d,b),(d,a)}



Directed Graph



Here,

V = {1, 2, 3, 4, 5, 6}

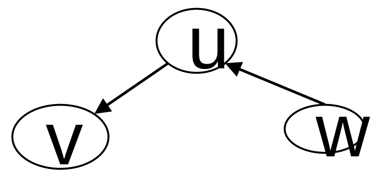
E = {(1, 2) (2, 5) (5, 6) (6, 3) (6, 4)}

Size and Degree

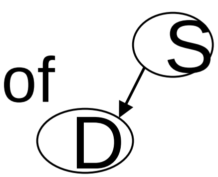
- The size of a graph is the number of *edges* in it
- The empty graph has no nodes and no edges
- If two nodes are connected by an edge, they are neighbors (and the nodes are adjacent to each other)
- The degree of a node is the number of edges it has, with loops counted twice

- For directed graphs,
  - In a directed graph vertex v is adjacent from u, if there is an edge leaving u and coming to v. and u is adjacent to v

v is adjacent from u.
u is adjacent from w



  - If a directed edge goes from node S to node D, we call S the source and D the destination of the edge
    - The edge is an out-edge of S and an in-edge of D

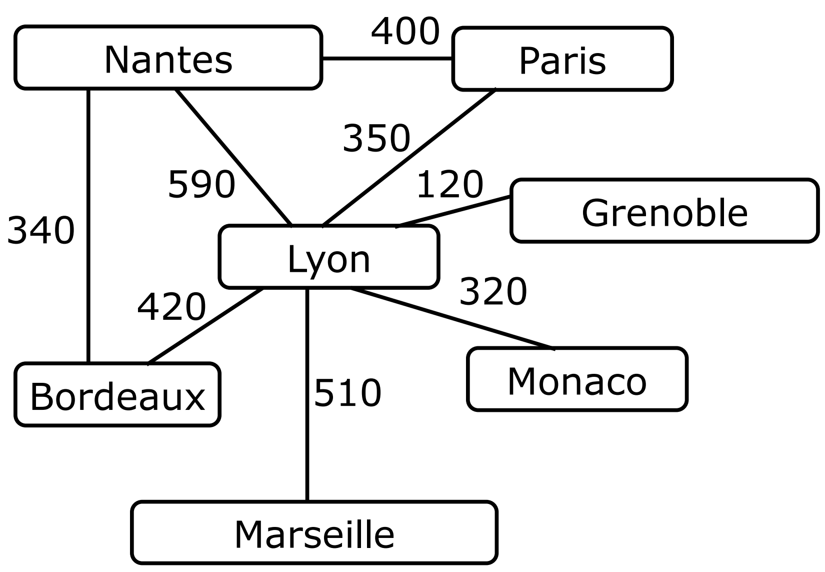


  - The in-degree of a node is the number of in-edges it has
  - The out-degree of a node is the number of out-edges it has

Path and Cycle

- In graph theory, a **path** in a graph is a sequence of edges which connect a sequence of vertices. Path has a first vertex, called its *start vertex*, and a last vertex, called its *end vertex*. Both of them are called *terminal vertices* of the path.

- A cycle is a path whose first and last nodes are the same



An undirected graph

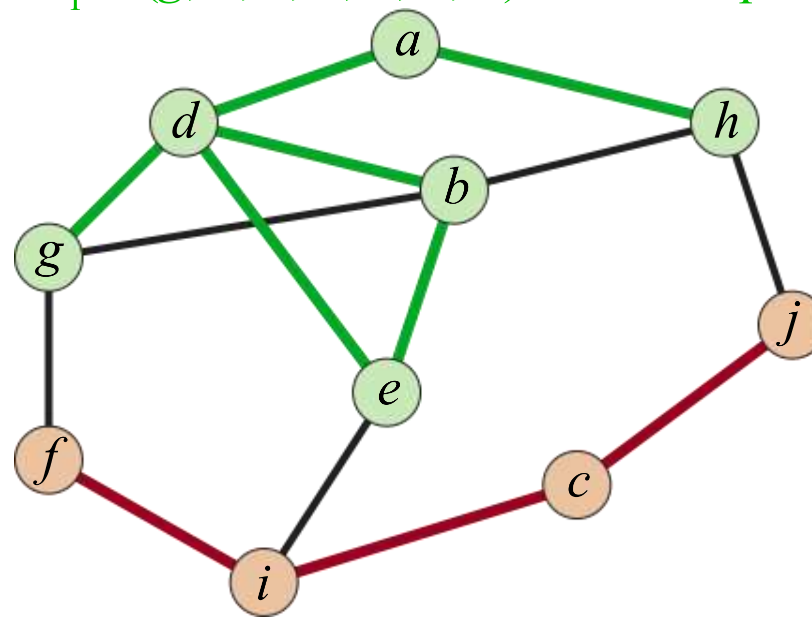
- Example: (Paris, Nantes, Bordeaux, Lyon) is a path
- Example: (Paris, Nantes, Lyon, Paris) is a cycle
- A cyclic graph contains at least one cycle
- An acyclic graph does not contain any cycles

Paths and Cycles

A ***path*** is a sequence of vertices
$P = (v_0, v_1, \ldots, v_k)$ such that,
for $1 \le i \le k$, edge $(v_{i-1}, v_i) \in E$.

Path P is ***simple*** if no vertex appears more than once in P.

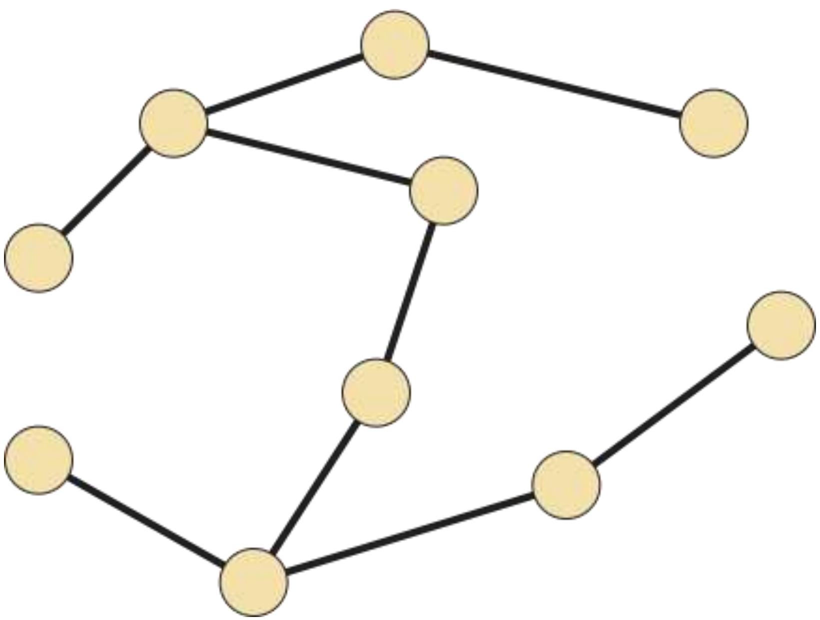
$P_1 = (g, d, e, b, d, a, h)$ is not simple.



$P_2 = (f, i, c, j)$ is simple.

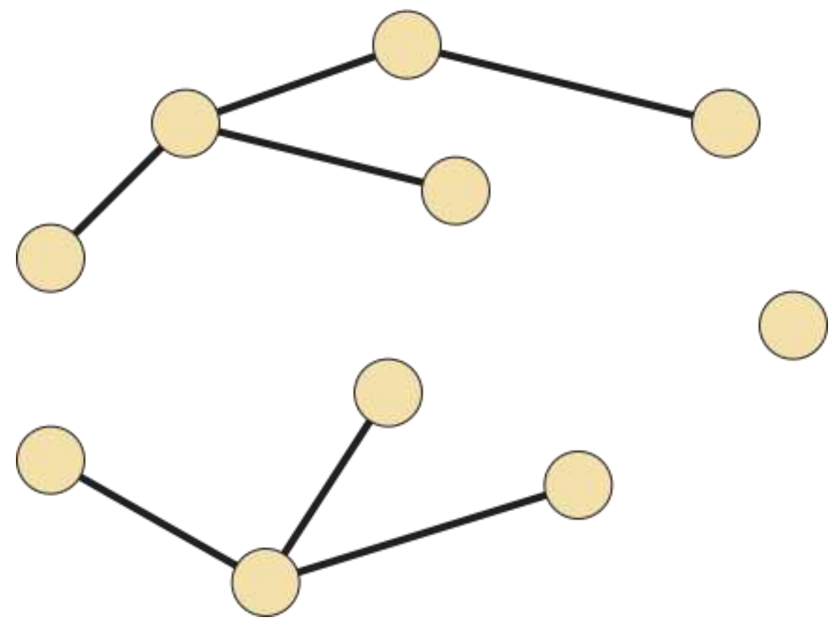
Trees and Forests

A ***tree*** is a graph that is connected and contains no cycles.



A ***forest*** is a graph that contains no cycles.

The connected components of a forest are trees.

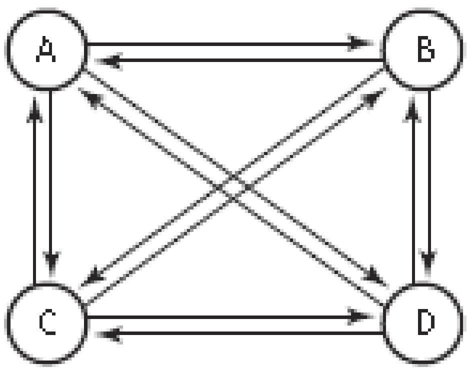


Graph Terminologies: Complete Graph

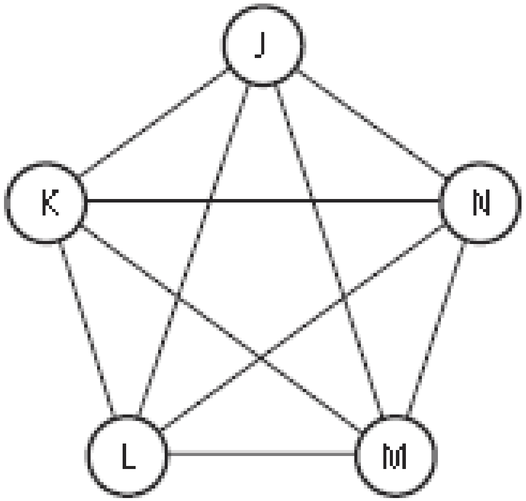
- **Complete graph** A graph in which every vertex is directly connected to every other vertex

- In a **complete graph**, every vertex is adjacent to every other vertex.

Example



(a) Complete directed graph.



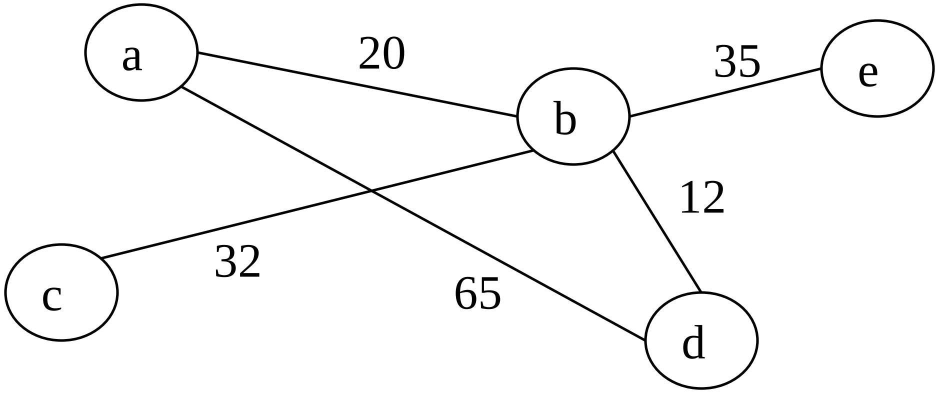
(b) Complete undirected graph.

N*(N-1) edges

N*(N-1)/2 edges

Weighted Graphs

- Graph G = (V,E) such that there are weights/costs associated with each edge
  - w((a,b)): cost of edge (a,b)



Applications of Graphs

- Google Map
- Facebook
- Computer networks
- Task of projects

Implementation of Graph

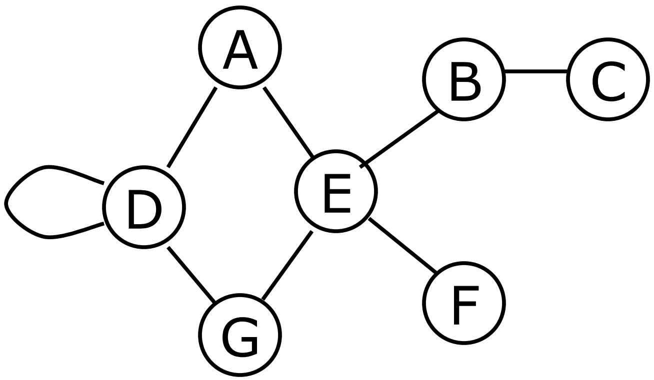
Graph Representation

- Adjacency-matrix Representation
- Adjacency Lists Representation

Representing Graphs: Adjacency Matrix

- Adjacency Matrix
  - Two dimensional matrix of size n x n where n is the number of vertices in the graph
  - a[i, j] = 0 if there is no edge between vertices i and j
  - a[i, j] = 1 if there is an edge between vertices i and j
  - Undirected graphs have both a[i, j] and a[j, i] = 1 if there is an edge between vertices i and j
  - a[i, j] = weight for weighted graphs

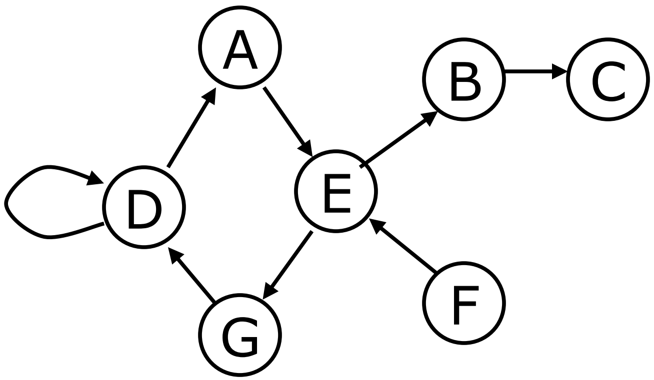
Adjacency-matrix representation I



	A	B	C	D	E	F	G
A				●	●		
B			●		●		
C		●					
D	●			●			●
E	●	●				●	●
F					●		
G				●	●		

- One simple way of representing a graph is the adjacency matrix
- A 2-D array has a mark at [i][j] if there is an edge from node i to node j
- The adjacency matrix is symmetric about the main diagonal
- This representation is only suitable for small graphs! (Why?)

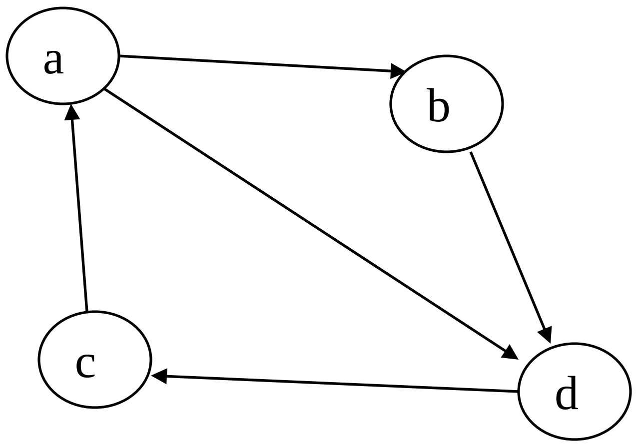
Adjacency-matrix representation II



	A	B	C	D	E	F	G
A					●		
B			●				
C							
D	●			●			
E		●					●
F					●		
G				●			

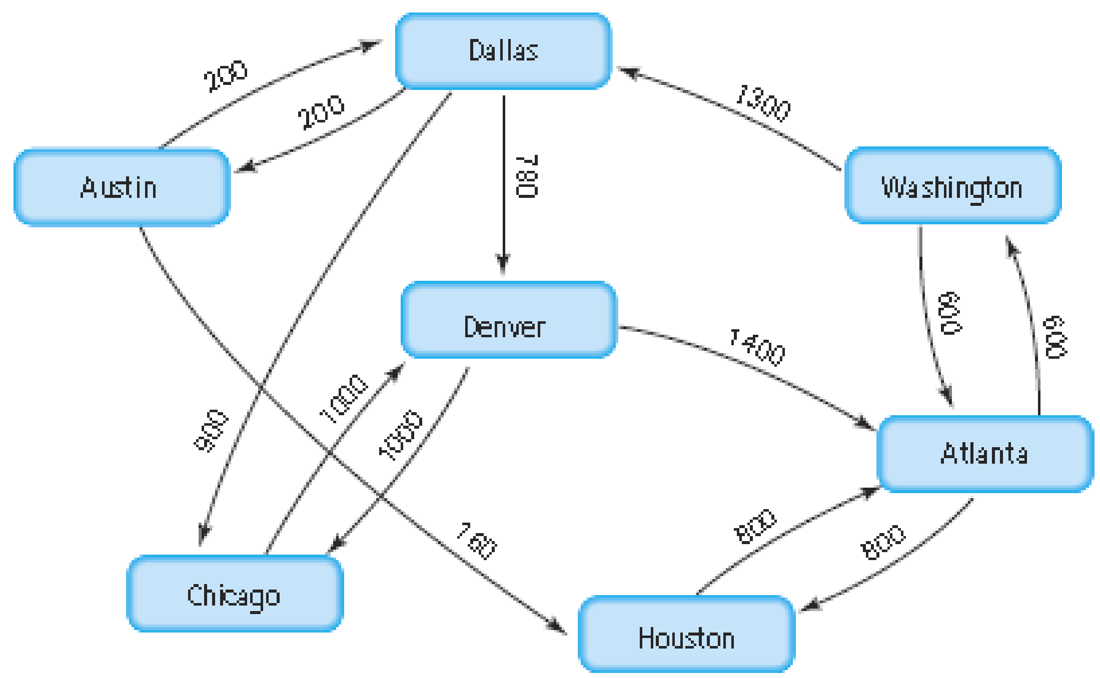
- An adjacency matrix can equally well be used for digraphs (directed graphs)
- A 2-D array has a mark at [i][j] if there is an edge from node i to node j
- Again, this is only suitable for *small* graphs!

Adjacency Matrix - Un weighted graph Example



	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	1	0	0	0
d	0	0	1	0

Adjacency Matrix – weighted graph Example



Adjacency Matrix - weighted graph Example

Austin
160 miles
Houston
800 miles
Atlanta
600 miles
Washington

Austin
200 miles
Dallas
780 miles
Denver
1400 miles
Atlanta
600 miles
Washington

Adjacency Matrix – weighted graph Example

.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"

.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]	0	0	0	0	0	800	600
[1]	0	0	0	200	0	160	0
[2]	0	0	0	0	1000	0	0
[3]	0	200	900	0	780	0	0
[4]	1400	0	1000	0	0	0	0
[5]	800	0	0	0	0	0	0
[6]	600	0	0	1300	0	0	0

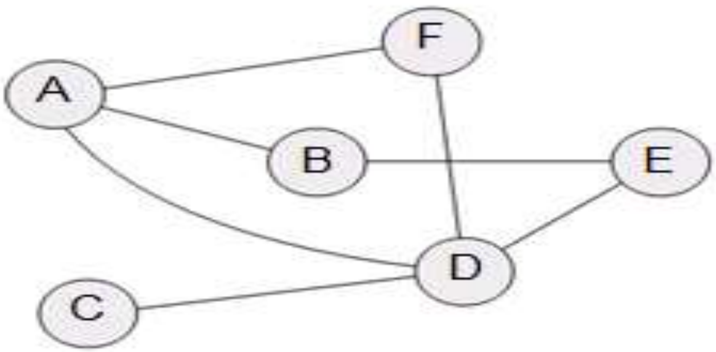
Representing Graphs: Adjacency List

- **Adjacency List**
  - Array of lists
  - Each vertex has an array entry
  - A vertex w is inserted in the list for vertex v if there is an outgoing edge from v to w

Adjacency Lists Representation

Graphs and Digraphs — Examples

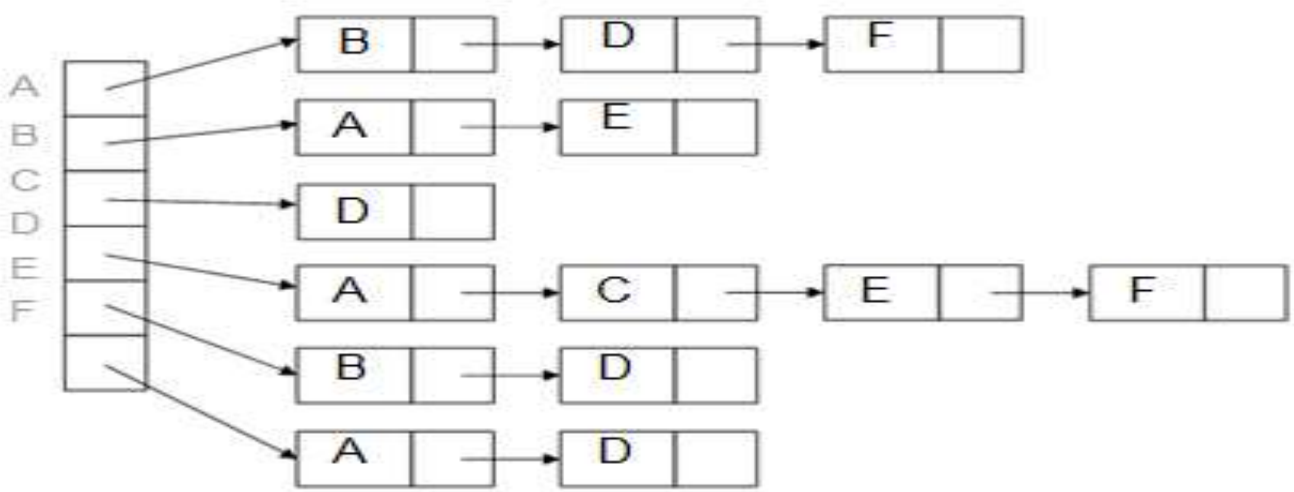
An (undirected) graph $G = (V,E)$



adjacency matrix for G

$$
\begin{array}{c|cccccc}
 & A & B & C & D & E & F \\
\hline
A & 0 & 1 & 0 & 1 & 0 & 1 \\
B & 1 & 0 & 0 & 0 & 1 & 0 \\
C & 0 & 0 & 0 & 1 & 0 & 0 \\
D & 1 & 0 & 1 & 0 & 1 & 1 \\
E & 0 & 1 & 0 & 1 & 0 & 0 \\
F & 1 & 0 & 0 & 1 & 0 & 0
\end{array}
$$

adjacency list for G



Adjacency Lists Representation

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent to node i
  - The nodes in the list L[i] are in no particular order
  - An adjacency list for a weighted graph should contain two elements in the list nodes – one element for the vertex and the second element for the weight of that edge

Pros and Cons of Adjacency Matrix

- Pros:
  - Simple to implement
  - Easy and fast to tell if a pair (i,j) is an edge: simply check if A[i][j] is 1 or 0
- Cons:
  - No matter how few edges the graph has, the matrix takes $O(n^2)$ in memory

Pros and Cons of Adjacency Lists

Pros:

- Saves on space (memory): the representation takes as many memory as there are nodes and edge.

Cons:

- It can take up to O(n) time to determine if a pair of nodes (i,j) is an edge: one would have to search the linked list L[i], which takes time proportional to the length of L[i].

Implementation

```
// A structure to represent an adjacency list node
struct AdjListNode
{
        int dest;
        struct AdjListNode* next;
};
// A structure to represent an adjacency list
struct AdjList
{
        struct AdjListNode *head;
};
```

```
// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
	int V;
	struct AdjList* array;
};
// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest)
{
	AdjListNode* newNode = new AdjListNode;
	newNode->dest = dest;
	newNode->next = NULL;
	return newNode;
}
```

```
// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{  // you can use new command as well
Graph* graph = new Graph;
        graph->V = V;

        // Create an array of adjacency lists. Size of array will be V
   graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

// Initialize each adjacency list as empty by making head as NULL
int i;
        for (i = 0; i < V; ++i)
                graph->array[i].head = NULL;

        return graph;
}
```

```
// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest)
{ // Add an edge from src to dest. A new node is added to the adjacency list of src. The node is added at the begining
        AdjListNode* newNode = newAdjListNode(dest);
        newNode->next = graph->array[src].head;
        graph->array[src].head = newNode;
// Since graph is undirected, add an edge from dest to src also
        newNode = newAdjListNode(src);
        newNode->next = graph->array[dest].head;
        graph->array[dest].head = newNode;
}
```

```
// A utility function to print the adjacency list representation of graph
void printGraph(struct Graph* graph)
{
        int v;
        for (v = 0; v < graph->V; ++v)
        {
        struct AdjListNode* pCrawl = graph->array[v].head;
        printf("\n Adjacency list of vertex %d\n head ", v);
                while (pCrawl)
                {
                        printf("-> %d", pCrawl->dest);
                        pCrawl = pCrawl->next;
                }
                printf("\n");
        }
}
```

```
int main()
{
        int V = 5;
        struct Graph* graph = createGraph(V);
        addEdge(graph, 0, 1);
        addEdge(graph, 0, 4);
        addEdge(graph, 1, 2);
        addEdge(graph, 1, 3);
        addEdge(graph, 1, 4);
        addEdge(graph, 2, 3);
        addEdge(graph, 3, 4);
        printGraph(graph);
        return 0;
}
```