## INTRODUCTION:

- Searching a graph means systematically following the edges of the graph so as to visit the vertices of the graph.
- A graph-searching algorithm can discover much about the structure of a graph.
- Techniques for searching a graph are at the heart of the field of graph algorithms.
  **Eg:** Many algorithms begin by searching their input graph to obtain this structural information. Other graph algorithms are organized as simple elaborations of basic graph-searching algorithms.

## NOTE:

- In describing the running time of a graph algorithm on a given graph G = (V, E), we usually measure the **size of the input in terms of the number of vertices |V| and the number of edges |E|** of the graph. This means that there are two relevant parameters describing the size of the input, not just one.

- Inside asymptotic notation (such as O-notation or -notation), and only inside such notation, the symbol V denotes |V| and the symbol E denotes |E|.
  Eg: For example, we might say, "the algorithm runs in time *O(V E)*," meaning that the algorithm runs in time *O(|V| |E|)*.

- In pseudocode, we denote the vertex set of a graph *G* by *V*[*G*] and its edge set by *E*[*G*].

## Representations of graphs

- There are **two standard ways** to represent a graph G = (V, E): as a collection of adjacency lists or as an adjacency matrix.

- Either way is applicable to both directed and undirected graphs.

- The adjacency-list representation is usually preferred, because it provides a compact way to represent ***sparse*** graphs (|E| is much less than $|V|^2$)

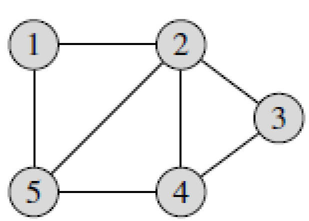
- An adjacency-matrix representation may be preferred, however, when the graph is dense ($|E|$ is close to $|V|^2$ OR when we need to be able to tell quickly if there is an edge connecting two given vertices.)
  **Eg:** For example, two of the all-pairs shortest-paths algorithms assume that their input graphs are represented by adjacency matrices

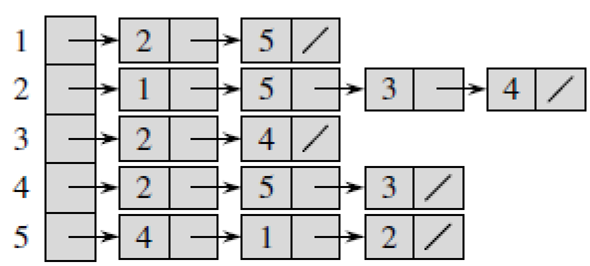
- Most of the graph algorithms presented here assume that an input graph is represented in **adjacency-list form.**

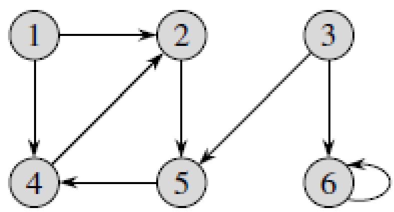
## The adjacency-list representation

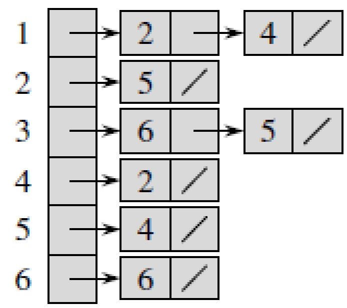
- The ***adjacency-list representation*** of a graph *G = (V, E)* consists of an array *Adj* of |*V*| lists, one for each vertex in V.
- For each u ∈ V, the adjacency list Adj[u] contains all the vertices v such that there is an edge (u, v) ∈ E. (That means Adj[u] consists of all the vertices adjacent to u in G)

- The vertices in each adjacency list are typically stored in an arbitrary order.







- If G is a directed graph, the sum of the lengths of all the adjacency lists is |E|, since an edge of the form (u, v) is represented by having v appear in Adj[u]. If G is an undirected graph, the sum of the lengths of all the adjacency lists is 2 |E|, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa.

- For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of **memory it requires is *Θ (V + E)***.

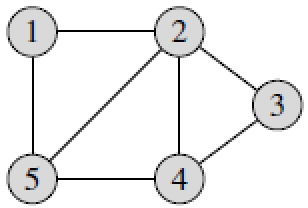
- A potential disadvantage of the adjacency-list representation is that there is no quicker way to determine if a given edge (u, v) is present in the graph than to search for v in the adjacency list Adj[u]. (This disadvantage can be remedied by an adjacency-matrix representation of the graph, at the cost of using asymptotically more memory.)

**<u>The adjacency-matrix representation</u>**

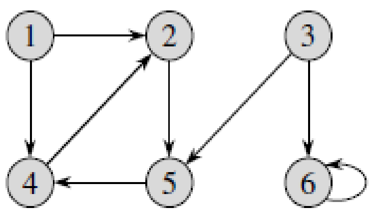
- For the adjacency-matrix representation of a graph G = (V, E), we assume that the vertices are numbered 1, 2, . . . , |V| in some **arbitrary** manner.

- Then the adjacency-matrix representation of a graph G consists of a |V| × |V| matrix A = ($a_{ij}$) such that

$$a_{ij} = \begin{cases} 1 & if\ (i,j) \in E \\ 0 & otherwise \end{cases}$$




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

- The adjacency matrix of a graph requires $\boldsymbol{\Theta(V^2)}$ memory, independent of the number of edges in the graph.
- Observe the symmetry along the main diagonal of the adjacency matrix in an undirected graph.
- Although the adjacency-list representation is asymptotically at least as efficient as the adjacency-matrix representation, the simplicity of an adjacency matrix may make it preferable when graphs are reasonably small.

**// simplified versions of DFS & BFS**

(Any particular graph algorithm may depend on the way G is maintained in memory. Here we assume G is maintained in memory by its adjacency structure.)

During the execution of our algorithms, each vertex (node) N of G will be in one of three states, called the status of N, as follows:
STATUS = 1: (Ready state) The initial state of the vertex N.
STATUS = 2: (Waiting state) The vertex N is on a (waiting) list, waiting to be processed.
STATUS = 3: (Processed state) The vertex N has been processed.

## DFS

**Algorithm 8.5 (Depth-first Search):** This algorithm executes a depth-first search on a graph $G$ beginning with a starting vertex $A$.

***Step 1.*** Initialize all vertices to the ready state (STATUS = 1).

***Step 2.*** Push the starting vertex $A$ onto STACK and change the status of $A$ to the waiting state (STATUS = 2).

***Step 3.*** Repeat Steps 4 and 5 until STACK is empty.

***Step 4.*** Pop the top vertex $N$ of STACK. Process $N$, and set STATUS $(N) = 3$, the processed state.

***Step 5.*** Examine each neighbor $J$ of $N$.

(*a*) If STATUS $(J) = 1$ (ready state), push $J$ onto STACK and reset STATUS $(J) = 2$ (waiting state).
(*b*) If STATUS $(J) = 2$ (waiting state), delete the previous $J$ from the STACK and push the current $J$ onto STACK.
(*c*) If STATUS $(J) = 3$ (processed state), ignore the vertex $J$.
[End of Step 3 loop.]

***Step 6.*** Exit.

## BFS

**Algorithm 8.6 (Breadth-first Search):** This algorithm executes a breadth-first search on a graph $G$ beginning with a starting vertex $A$.

***Step 1.*** Initialize all vertices to the ready state (STATUS = 1).

***Step 2.*** Put the starting vertex $A$ in QUEUE and change the status of $A$ to the waiting state (STATUS = 2).

***Step 3.*** Repeat Steps 4 and 5 until QUEUE is empty.

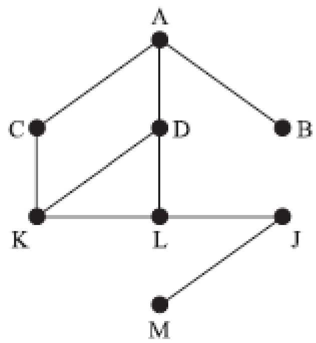
***Step 4.*** Remove the front vertex $N$ of QUEUE. Process $N$, and set STATUS $(N) = 3$, the processed state.

***Step 5.*** Examine each neighbor $J$ of $N$.
(*a*) If STATUS $(J) = 1$ (ready state), add $J$ to the rear of QUEUE and reset STATUS $(J) = 2$ (waiting state).
(*b*) If STATUS $(J) = 2$ (waiting state) or STATUS $(J) = 3$ (processed state), ignore the vertex $J$.
[End of Step 3 loop.]

***Step 6.*** Exit.

**Example:** Suppose the DFS Algorithm is applied to the graph in shown below, The vertices will be processed in the following order:

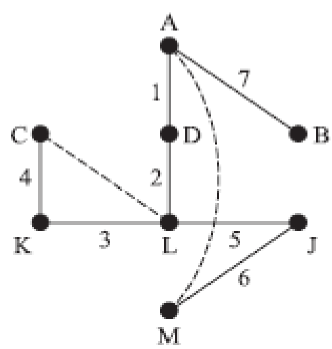
A, D, L, K, C, J, M, B




| Vertex | Adjacency list |
| --- | --- |
| A | B, C, D |
| B | A |
| C | A, K |
| D | A, K, L |
| J | L, M |
| K | C, D, L |
| L | D, J, K |
| M | J |

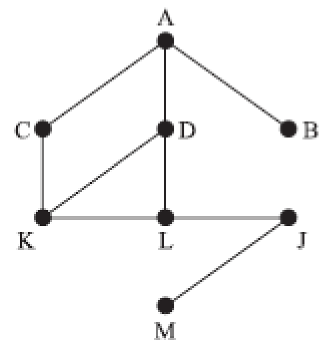
**<u>Sol$^n$</u>**

| STACK | Vertex |
| --- | --- |
| A | A |
| D, C, B | D |
| L, K, C, B | L |
| K, J, $\cancel{K}$, C, B | K |
| C, J, $\cancel{C}$, B | C |
| J, B | J |
| M, B | M |
| B | B |
| ∅ | |



**Example:** Suppose the BFS Algorithm is applied to the graph in shown below, The vertices will be processed in the following order:

A, B, C, D, K, L, J, M




| Vertex | Adjacency list |
|---|---|
| A | B, C, D |
| B | A |
| C | A, K |
| D | A, K, L |
| J | L, M |
| K | C, D, L |
| L | D, J, K |
| M | J |

**<u>Sol$^{n}$</u>**

| QUEUE | Vertex |
|---|---|
| A | A |
| D, C, B | B |
| D, C | C |
| D | D |
| L, K | K |
| L | L |
| J | J |
| M | M |
| ∅ | |