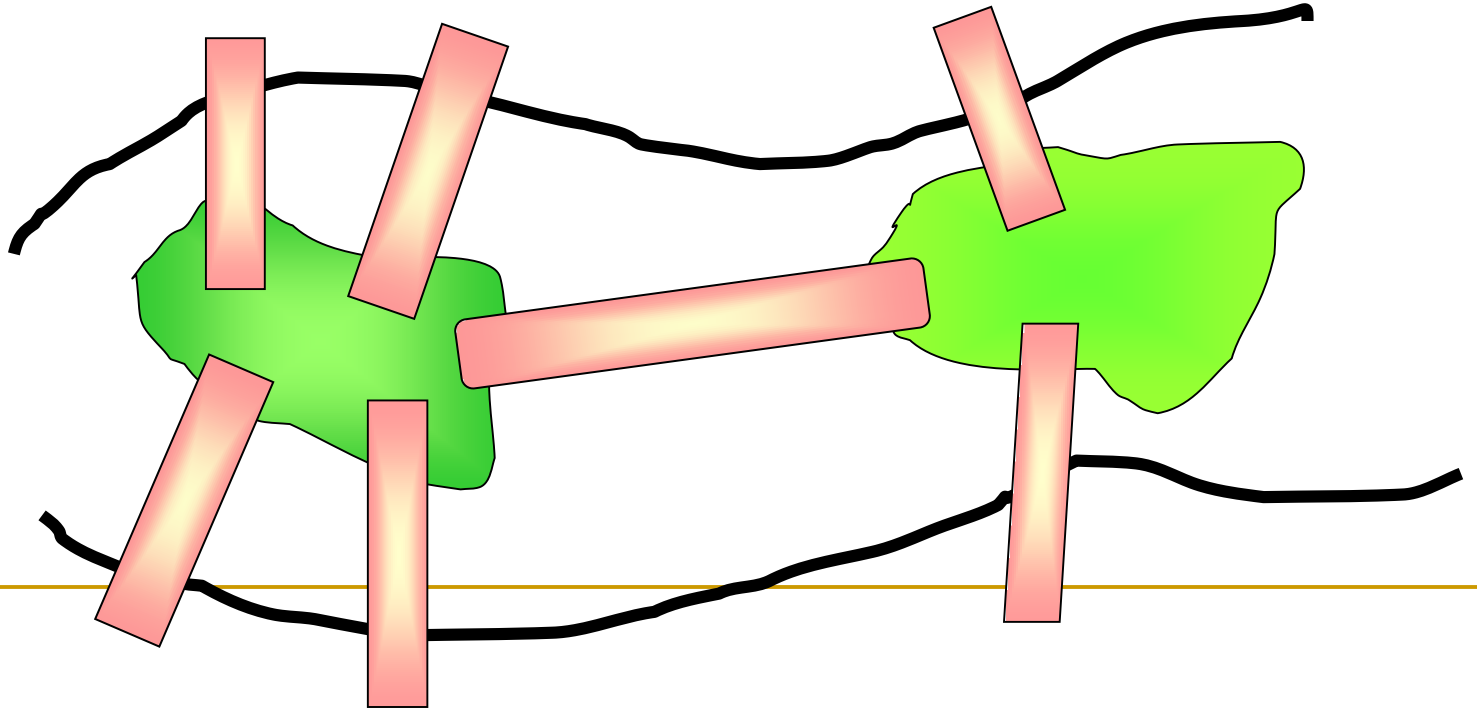
Graph (I)

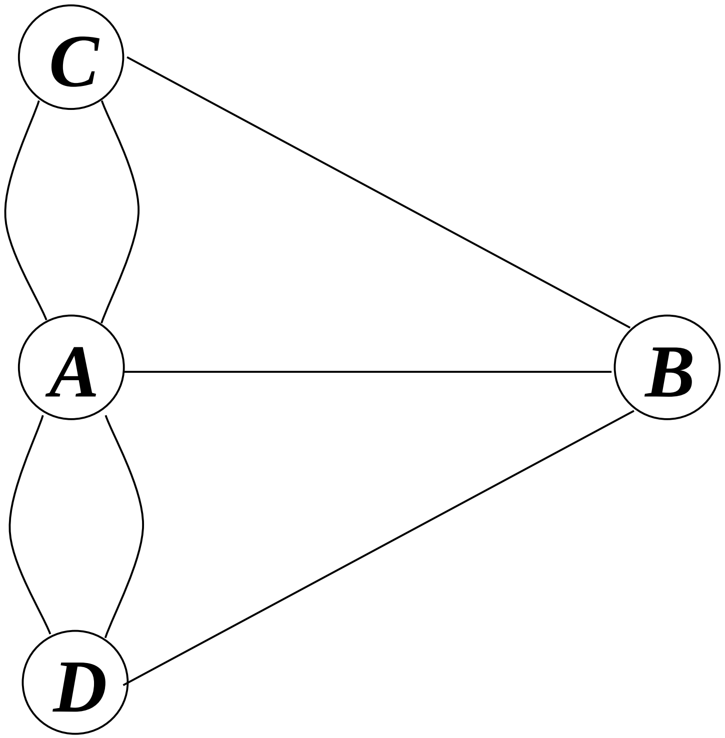
Bruce Nan
Email: xiaomingnan@gmail.com

Seven Bridges of Königsberg

- The **Seven Bridges of Königsberg** is a notable historical problem in mathematics. The problem was to find a walk through the city that would cross each bridge once and only once. The islands could not be reached by any route other than the bridges, and every bridge must have been crossed completely every time.



Euler's Analysis



- During any walk in the graph, the number of times one enters a non-terminal vertex equals the number of times one leaves it.
- If every bridge is traversed exactly once it follows that for each land mass (except possibly for the ones chosen for the start and finish), the number of bridges touching that land mass is **even**.
- However, all the four land masses in the original problem are touched by an **odd** number of bridges (one is touched by 5 bridges and the other three by 3). Since at most two land masses can serve as the endpoints of a putative walk, the existence of a walk traversing each bridge once leads to a contradiction.

Eulerian path

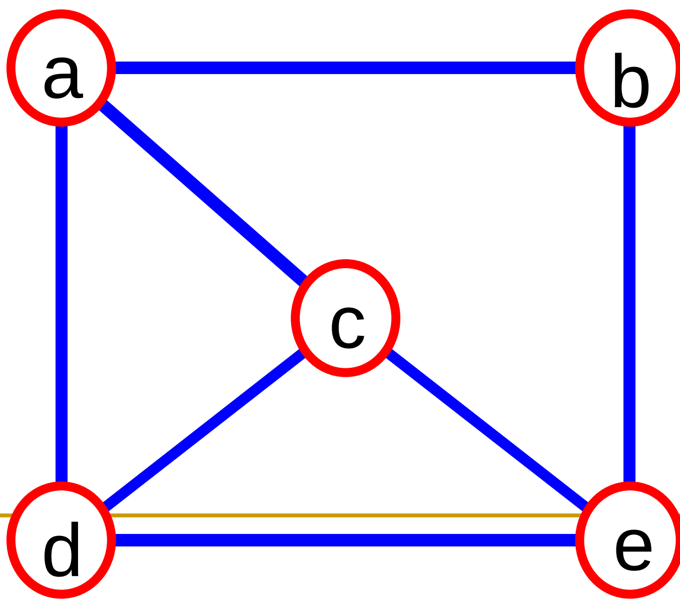
- In graph theory, an **Eulerian trail** is a trail in a graph which visits every edge exactly once. Similarly, an **Eulerian circuit** or **Eulerian cycle** is a Eulerian trail which starts and ends on the same vertex.

What is a Graph?

- A graph G = (V,E) is composed of:

  V: set of vertices

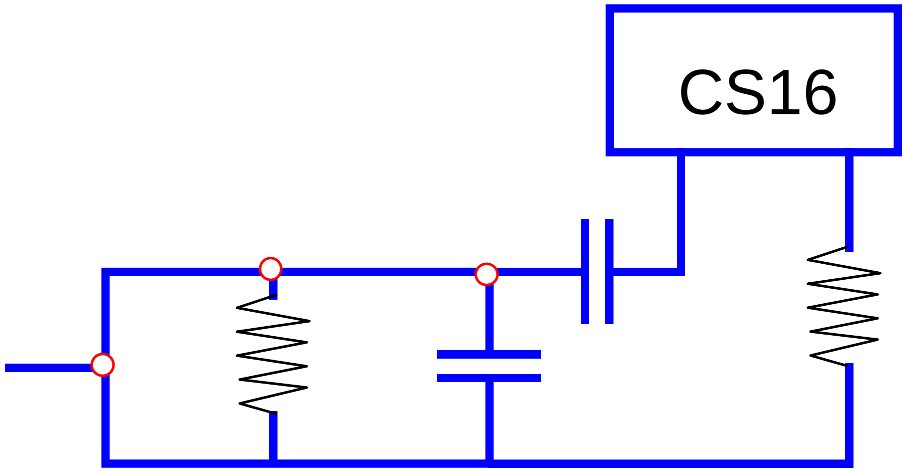
  E: set of edges connecting the vertices in V
- An edge e = (u,v) is a pair of vertices. Edges are sometimes referred to as arcs.
- Example



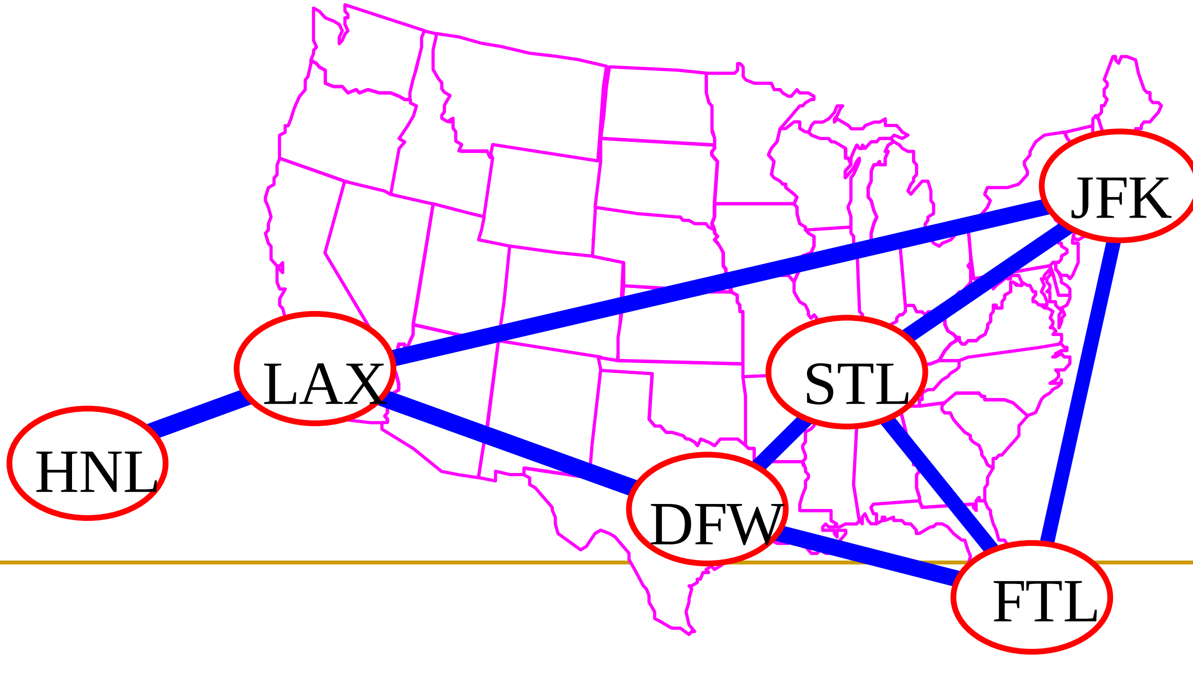
V= {a,b,c,d,e}

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

Applications

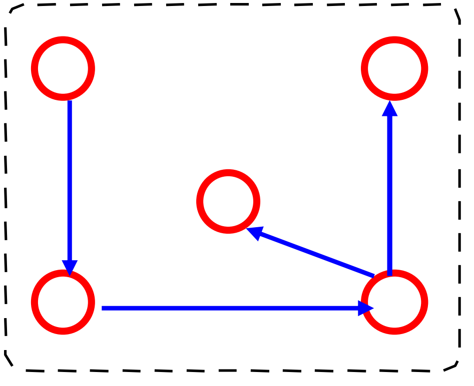


- electronic circuits
- networks (roads, flights, communications)



Directed Vs. Undirected Graph

- An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A directed graph is one in which each edge is a directed pair of vertices, $<v_0, v_1> \; != \; <v_1, v_0>$



Terminology: Adjacent

- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

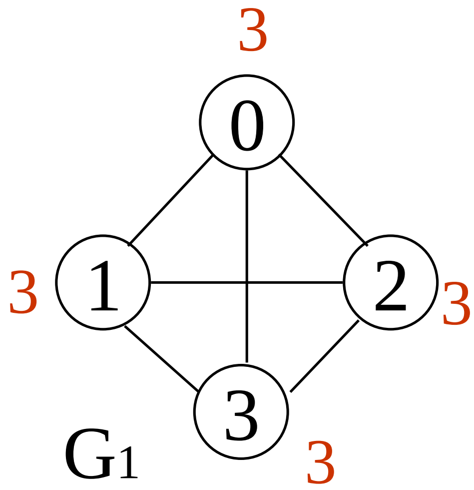
Degree of a Vertex

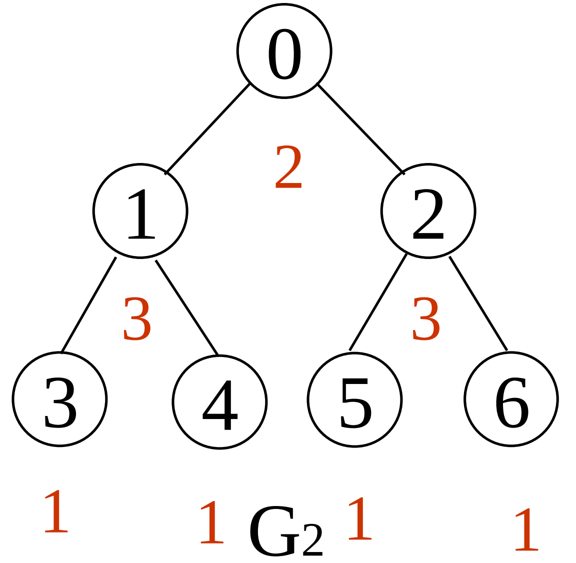
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

Why? Since adjacent vertices each count the adjoining edge, it will be counted twice

Examples

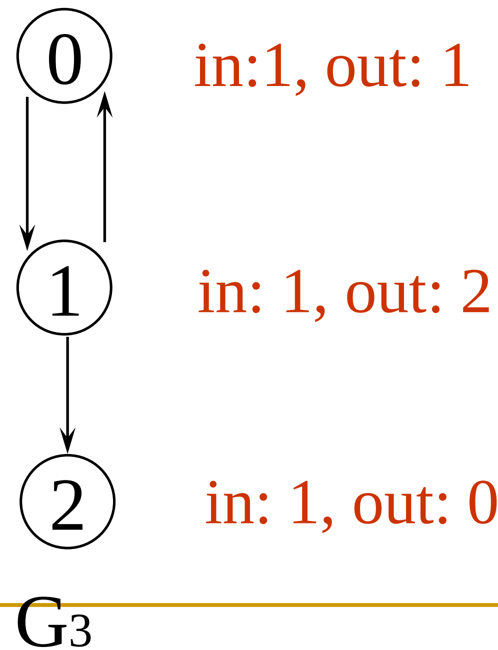




directed graph

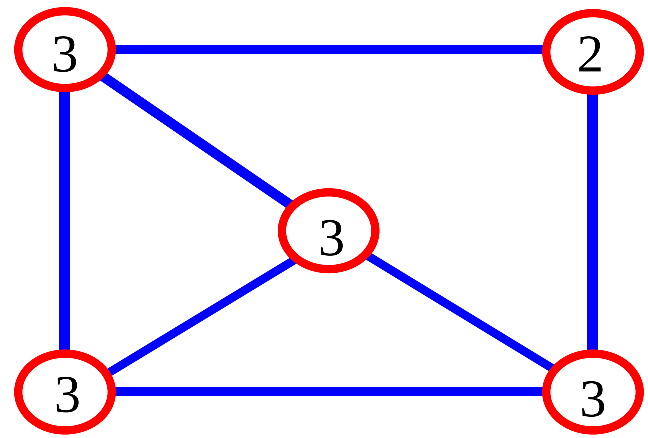
in-degree

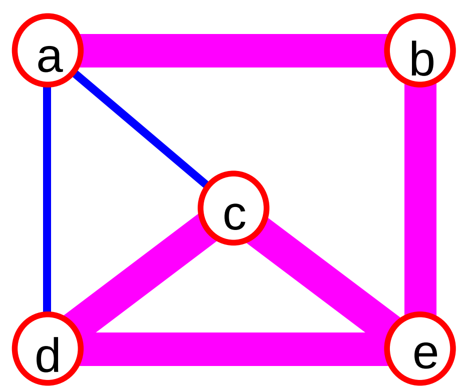
out-degree



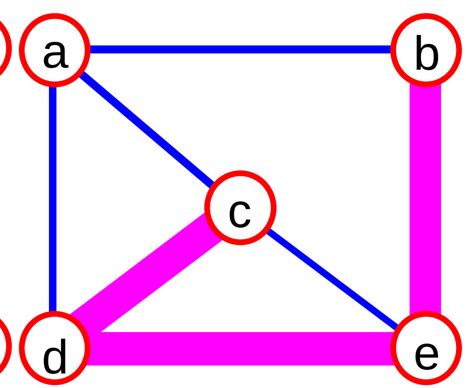
Terminology: Path

- path: sequence of vertices $v_1, v_2, . . . v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent. The length of such a path is the number of edges on the path, which is equal to k-1. We allow a path from a vertex to itself; if this path contains no edges, then the path lenght is 0.





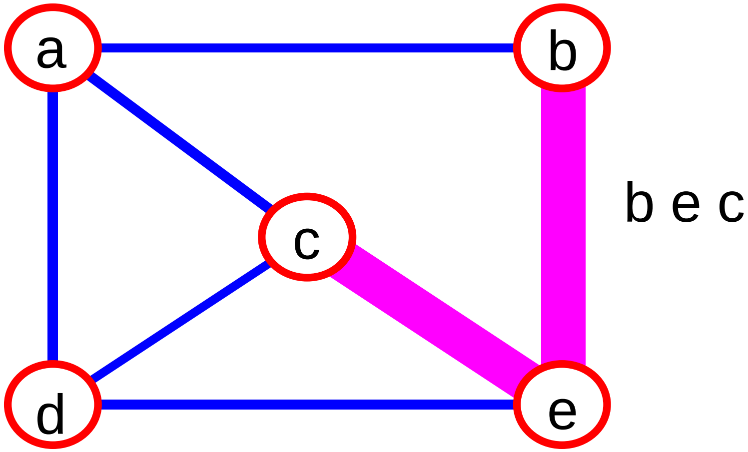
a b e d c



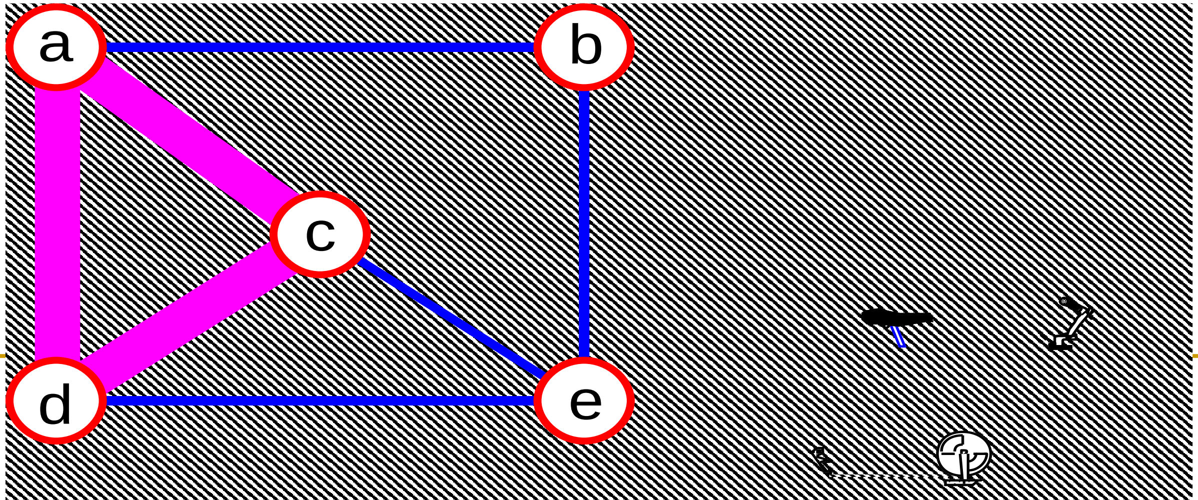
b e d c

More Terminology

- simple path: no repeated vertices

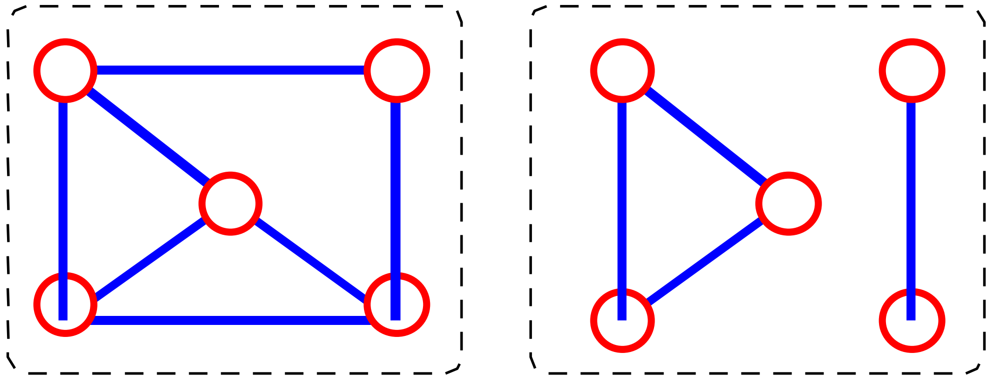


- cycle: simple path, except that the last vertex is the same as the first vertex



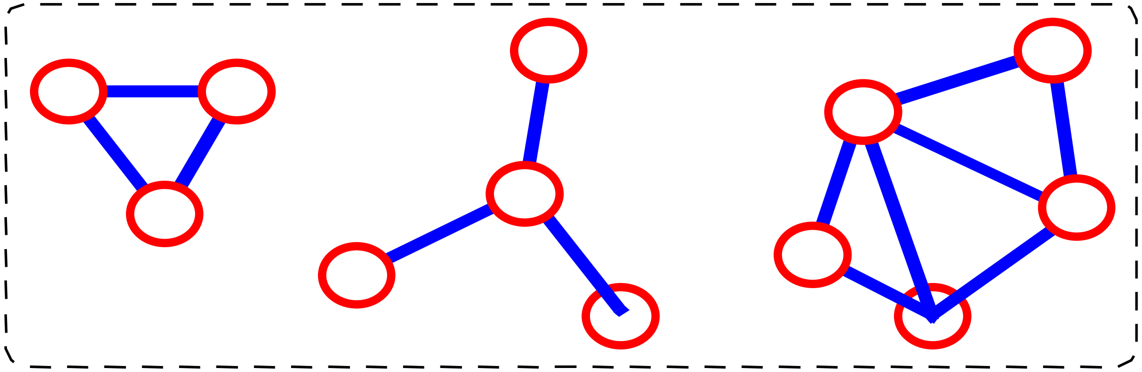
More Terminology

- An undirected graph is connected if there is a path from every vertex to every other vertex.
- A directed graph with this property is called strongly connected.

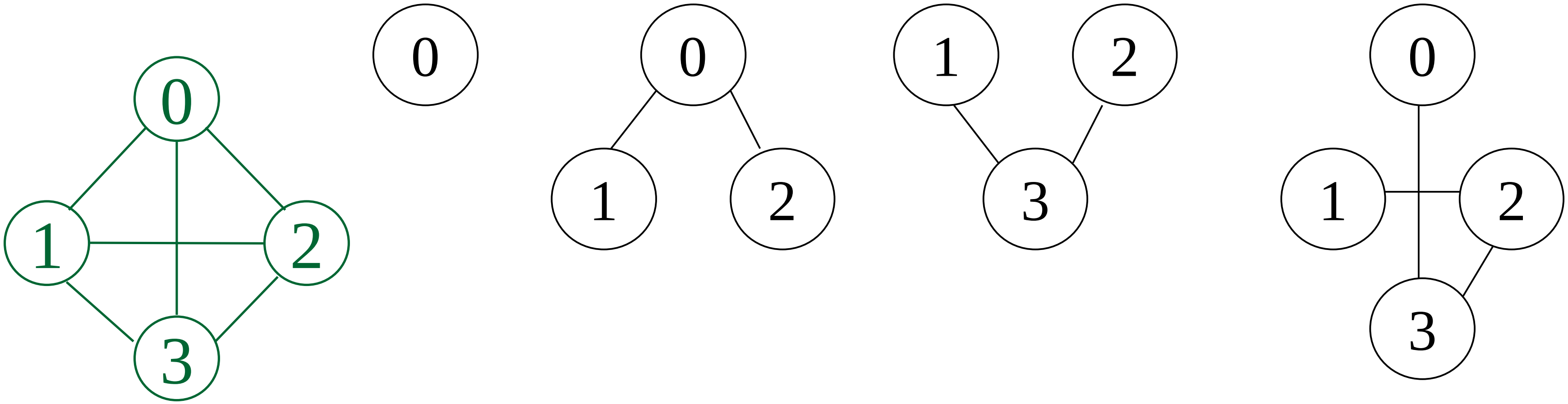


More Terminology

- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.



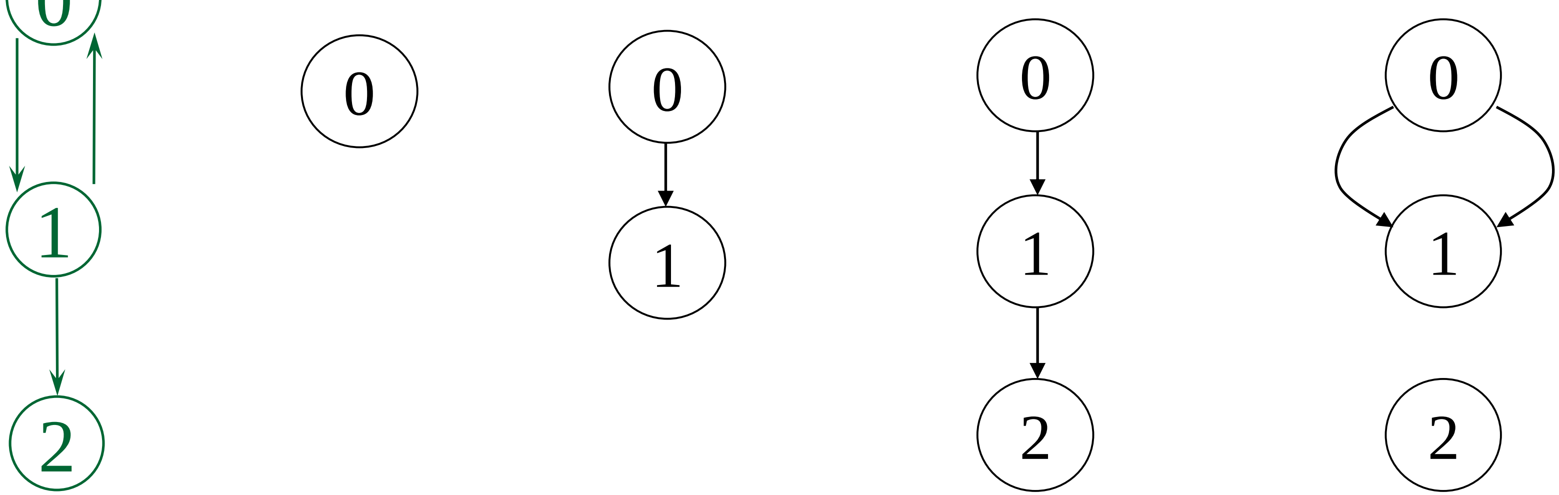
Examples



G_1

(i) (ii) (iii) (iv)

(a) Some of the subgraph of G_1



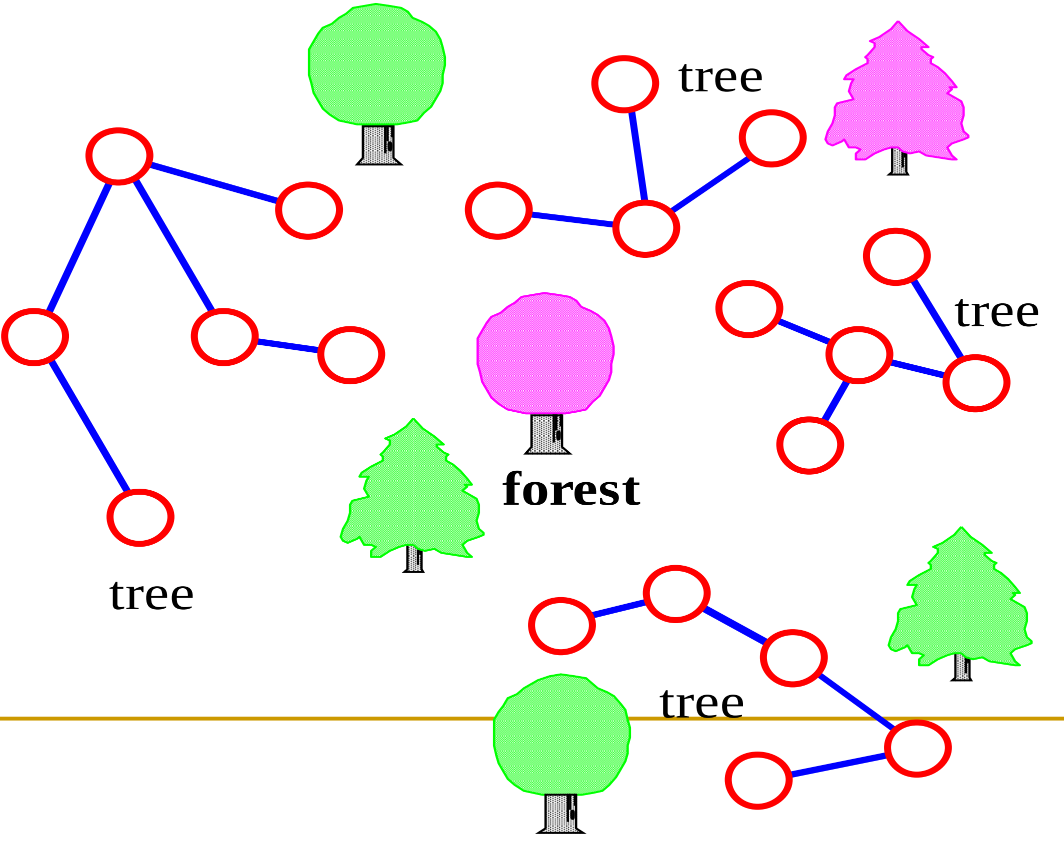
(i) (ii) (iii) (iv)

(b) Some of the subgraph of G_3

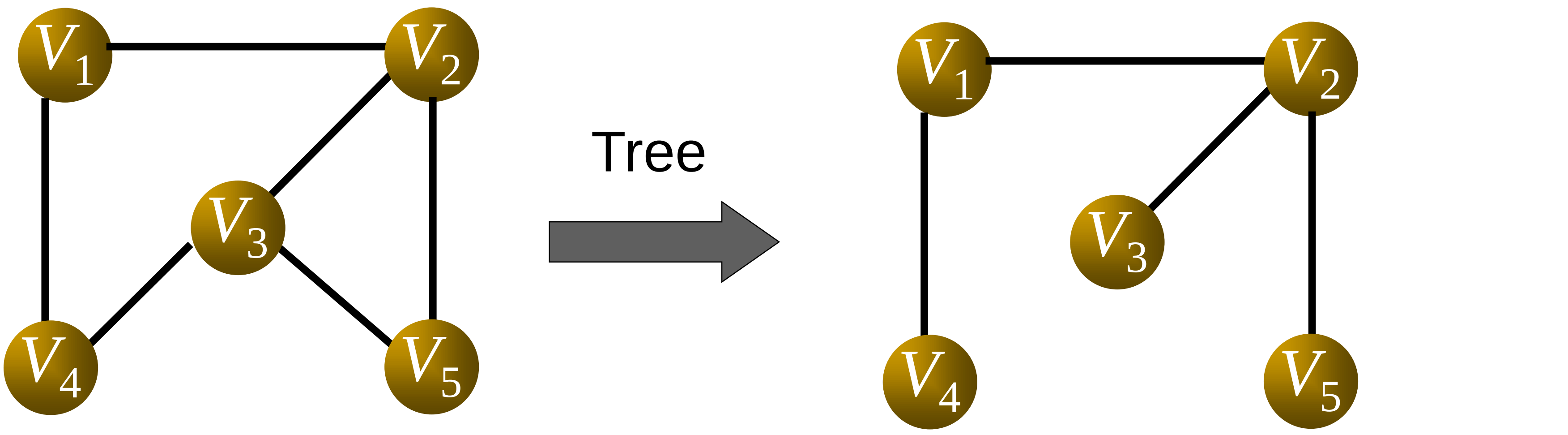
G_3

Tree and Forests

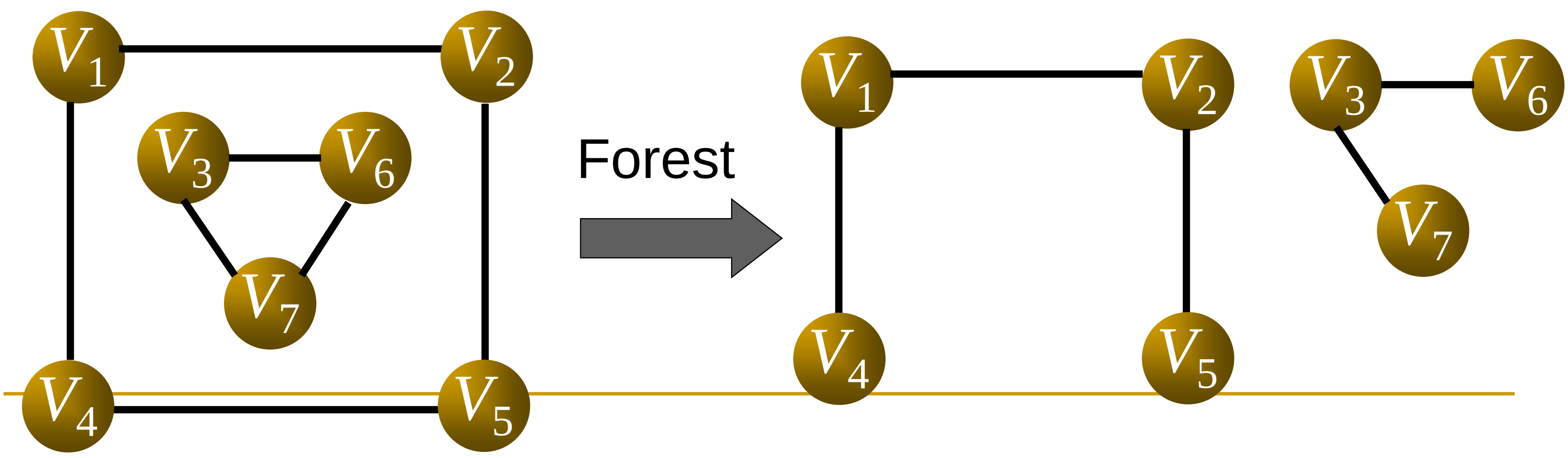
- tree - connected graph without cycles
- forest - collection of trees



Examples



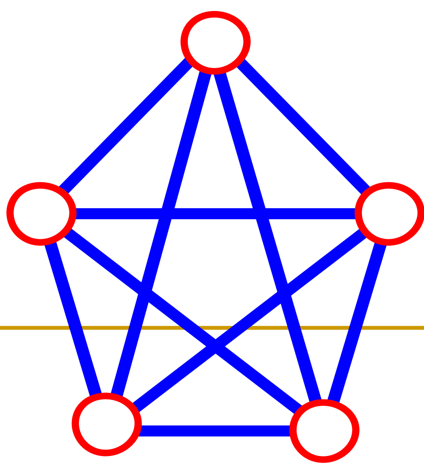
V_1
V_2
V_3
V_4
V_5
Tree
V_1
V_2
V_3
V_4
V_5



V_1
V_2
V_3
V_6
V_7
V_4
V_5
Forest
V_1
V_2
V_3
V_6
V_7
V_4
V_5

Connectivity

- Let **n** = #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n**-1 edges, however, we would have counted each edge twice! Therefore, intuitively, m = **n**(**n** -1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



$\mathbf{n} = 5$

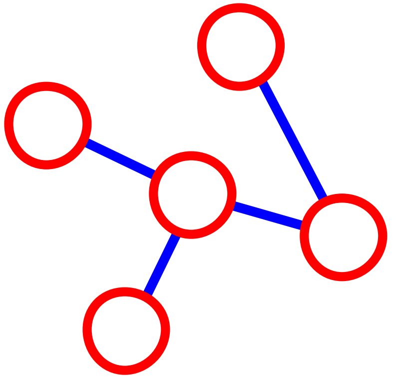
$\mathbf{m} = (5 * 4)/2 = 10$

More Connectivity

n = #vertices

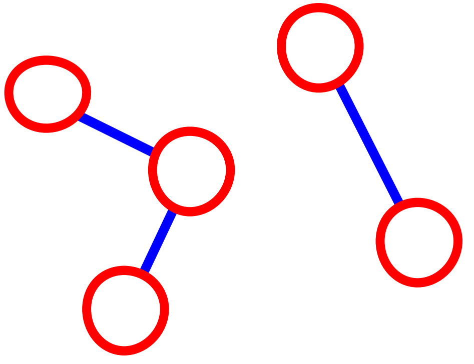
m = #edges

- For a tree m = n - 1



n = 5
m = 4

If m < n - 1, G is not connected



n = 5
m = 3

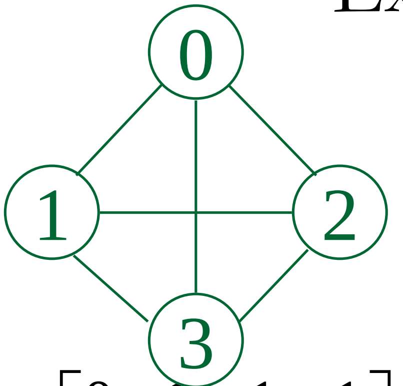
How to store a Graph?

- Adjacent Matrix
- Adjacent Lists

Adjacent Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix

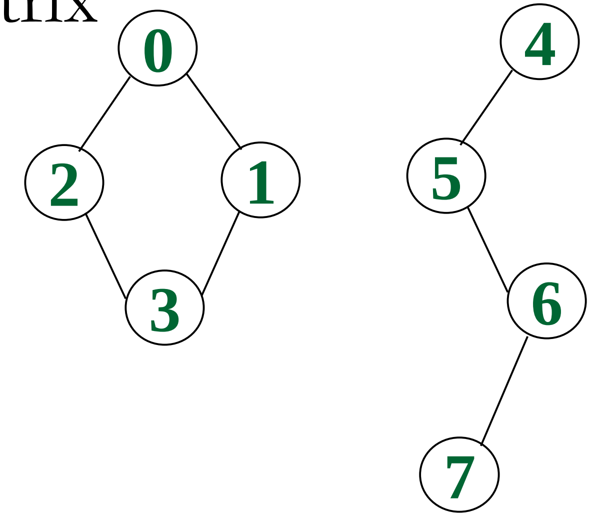


$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2



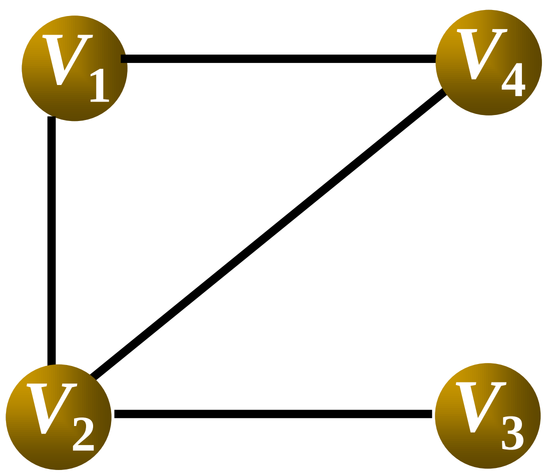
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

symmetric

More Adjacent Matrix

- How to get the degree of vertex i?



vertex=

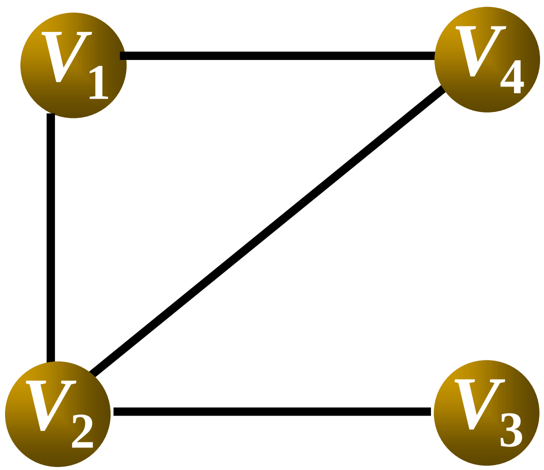
V_1	V_2	V_3	V_4

$$\textbf{arc=}\quad \begin{array}{cccc|c} V_1 & V_2 & V_3 & V_4 & \\ \end{array}$$

$$\textbf{arc=}\left(\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{array}\right)\begin{array}{c} V_1 \\ V_2 \\ V_3 \\ V_4 \end{array}$$

The number of non-zero data in i^{th} row

More Adjacent Matrix



vertex=

V_1	V_2	V_3	V_4

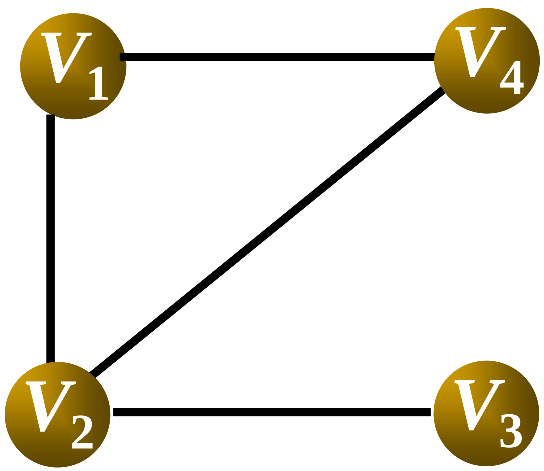
arc=

$$\begin{array}{cc} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} & \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} & \begin{array}{c} V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \end{array}$$

How to ensure if there is edge on v_i and v_j?

Test whether arc[i][j] is 1

More Adjacent Matrix



vertex=

V_1	V_2	V_3	V_4

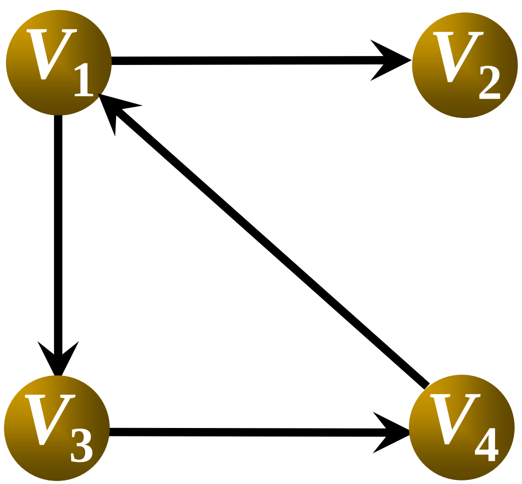
arc=

$$\begin{array}{cc} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} & \\ \left(\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{array}\right) & \begin{array}{c} V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \end{array}$$

How to find all adjacent vertices of v_i?

Traverse all i^{th} row elements, if arc[i][j] is 1, v_j is the adjacent vetex of v_i

Digraph Adjacent Matrix



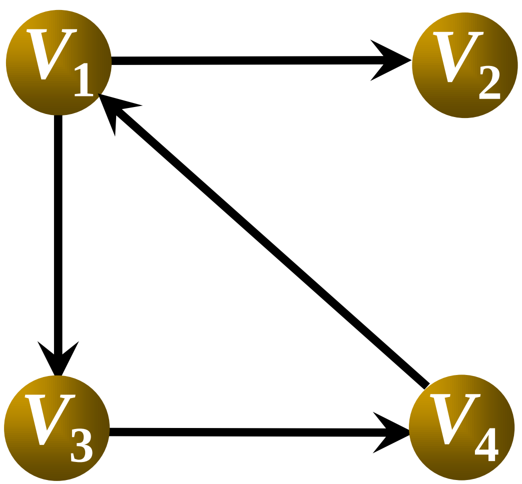
vertex= $\boxed{V_1 \mid V_2 \mid V_3 \mid V_4}$

$$
\textbf{arc=}\ \begin{array}{cc}
\begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} & \\
\left(\begin{array}{cccc} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{array}\right) & \begin{array}{c} V_1 \\ V_2 \\ V_3 \\ V_4 \end{array}
\end{array}
$$

How to find the out degree of v_i?

sum all elements on i[th] row

Digraph Adjacent Matrix



vertex=

V_1	V_2	V_3	V_4

$$\text{arc}=\begin{array}{cc} \begin{matrix} V_1 & V_2 & V_3 & V_4 \end{matrix} & \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} & \begin{matrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{matrix} \end{array}$$

How to find the in degree of v_i?

sum all elements on i^{th} column

More Adjacent Matrix

- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$

- For a digraph (= directed graph), the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

Create Adjacent Matrix

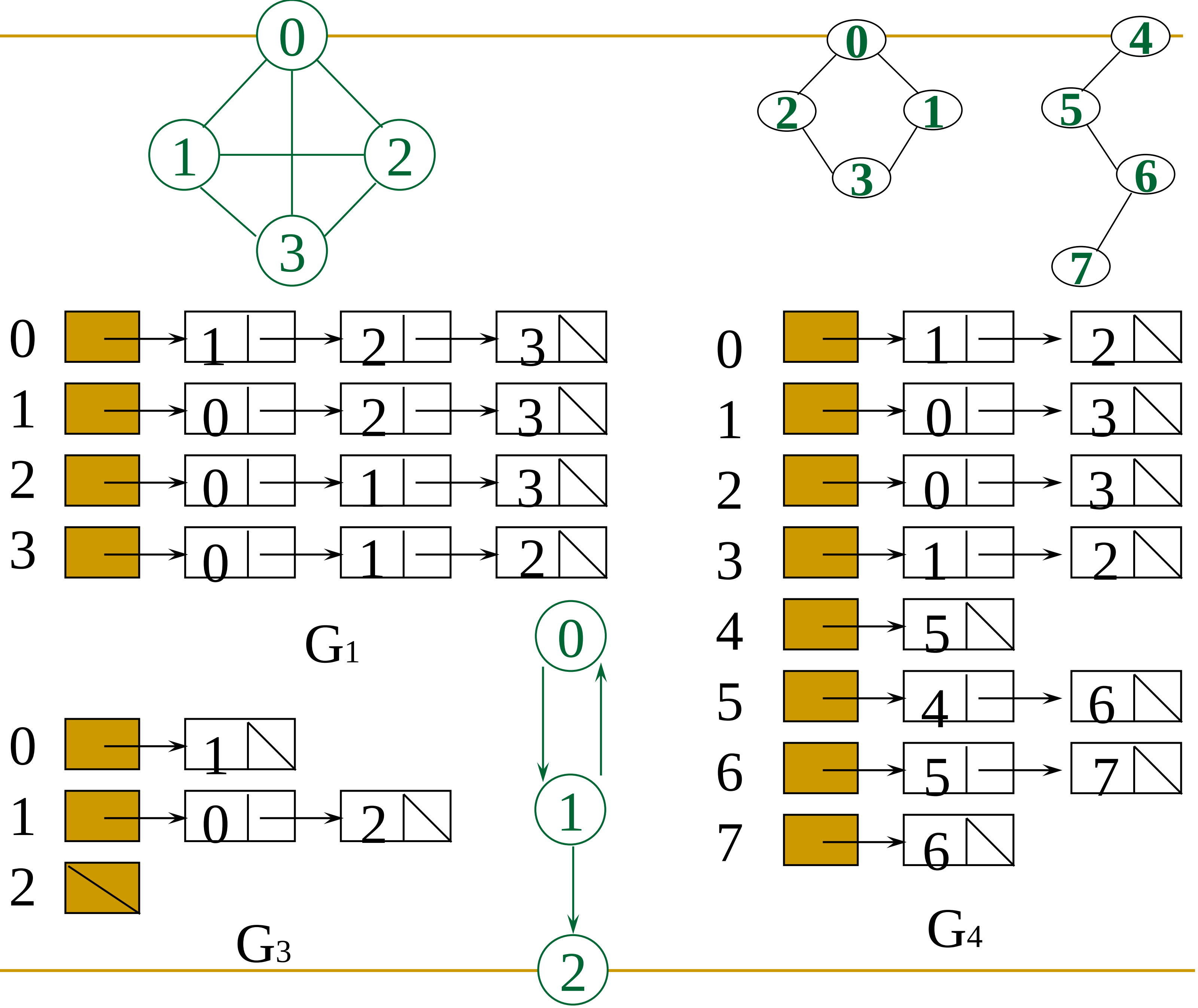
```
buildGraph(int a[ ], int n, int e)
{
   vertexNum=n; arcNum=e;
   for (i=0; i<vertexNum; i++)
      vertex[i]=a[i];
   for (i=0; i<vertexNum; i++)    //initialization
      for (j=0; j<vertexNum; j++)
         arc[i][j]=0;
   for (k=0; k<arcNum; k++)     //get each edge
   {
      cin>>i>>j;     //the vertices of the edge
      arc[i][j]=1;  arc[j][i]=1;  //adjacency
   }
}
```

Practice Time

- Implement following functions
  - Given a graph, create the adjacent matrix
  - Insert edges between u and v
  - Delete edge between u and v
  - Output the degree of u
  - Output all the adjacent vertices of u

Adjacency Lists

- The space complexity of Adjacent Matrix?
- Idea for Adjacency List:
  - For each vertex vi, using a list to store all adjacent vertices with vi, which is referred as vi’s edge list
- Implementation
  - Vector array



An undirected graph with n vertices and e edges ==> n head nodes and 2e list node

More Adjacency List

- degree of a vertex in an undirected graph
  - # of nodes in adjacency list
- # of edges in a graph
  - determined in O(n+e)
- out-degree of a vertex in a directed graph
  - # of nodes in its adjacency list
- in-degree of a vertex in a directed graph
  - traverse the whole data structure

Graph Traversal

- Problem: Search for a certain node or traverse all nodes in the graph
- Depth First Search
  - Once a possible path is found, continue the search until the end of the path
- Breadth First Search
  - Start several paths at a time, and advance in each one step at a time
- Topological Sorting

Graph Traversal

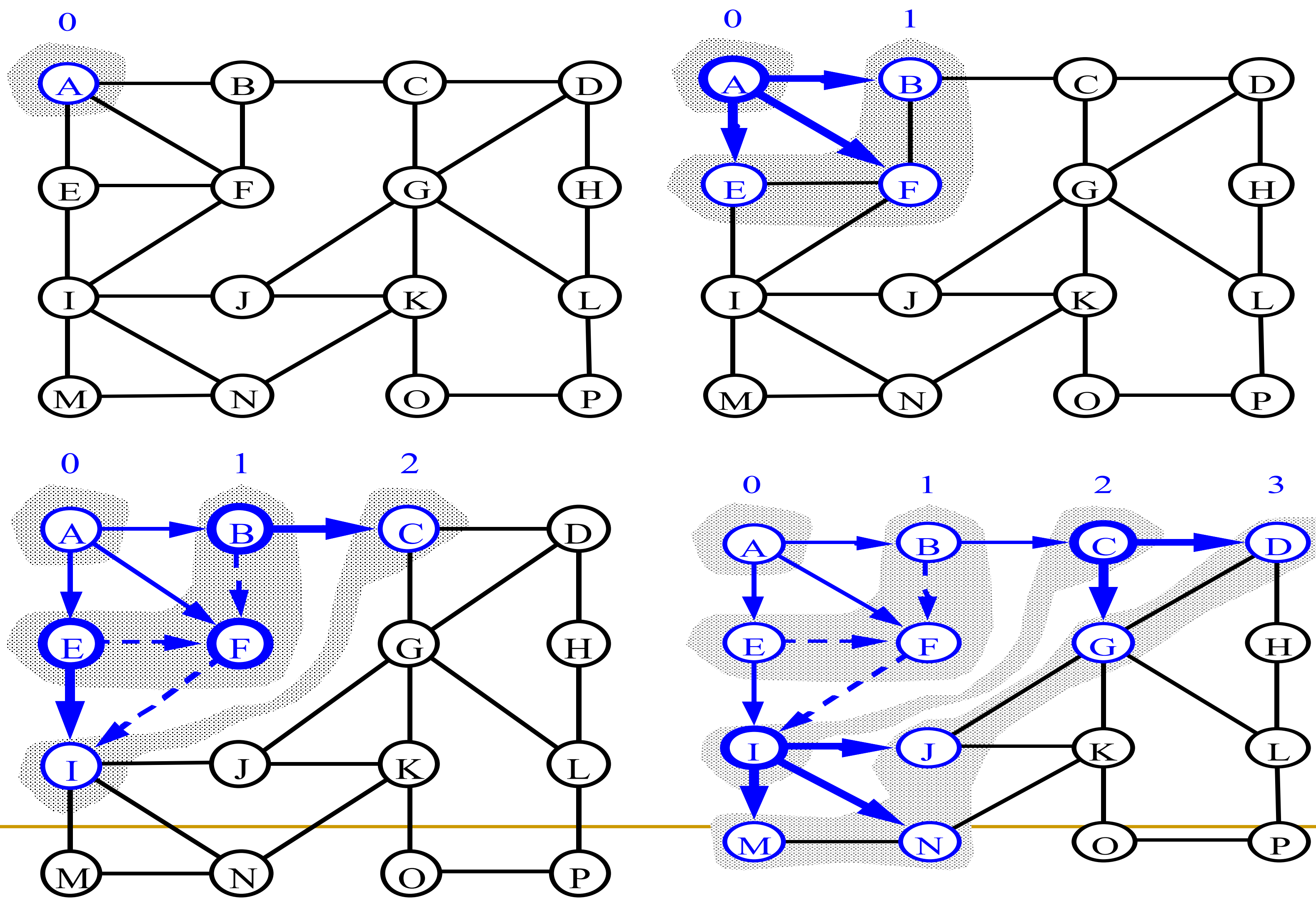
- Both graph traversal procedures share one fundamental idea, namely, that it is necessary to mark the vertices we have seen before so we don’t try to explore them again. Otherwise we get trapped in a maze and can’t find our way out.

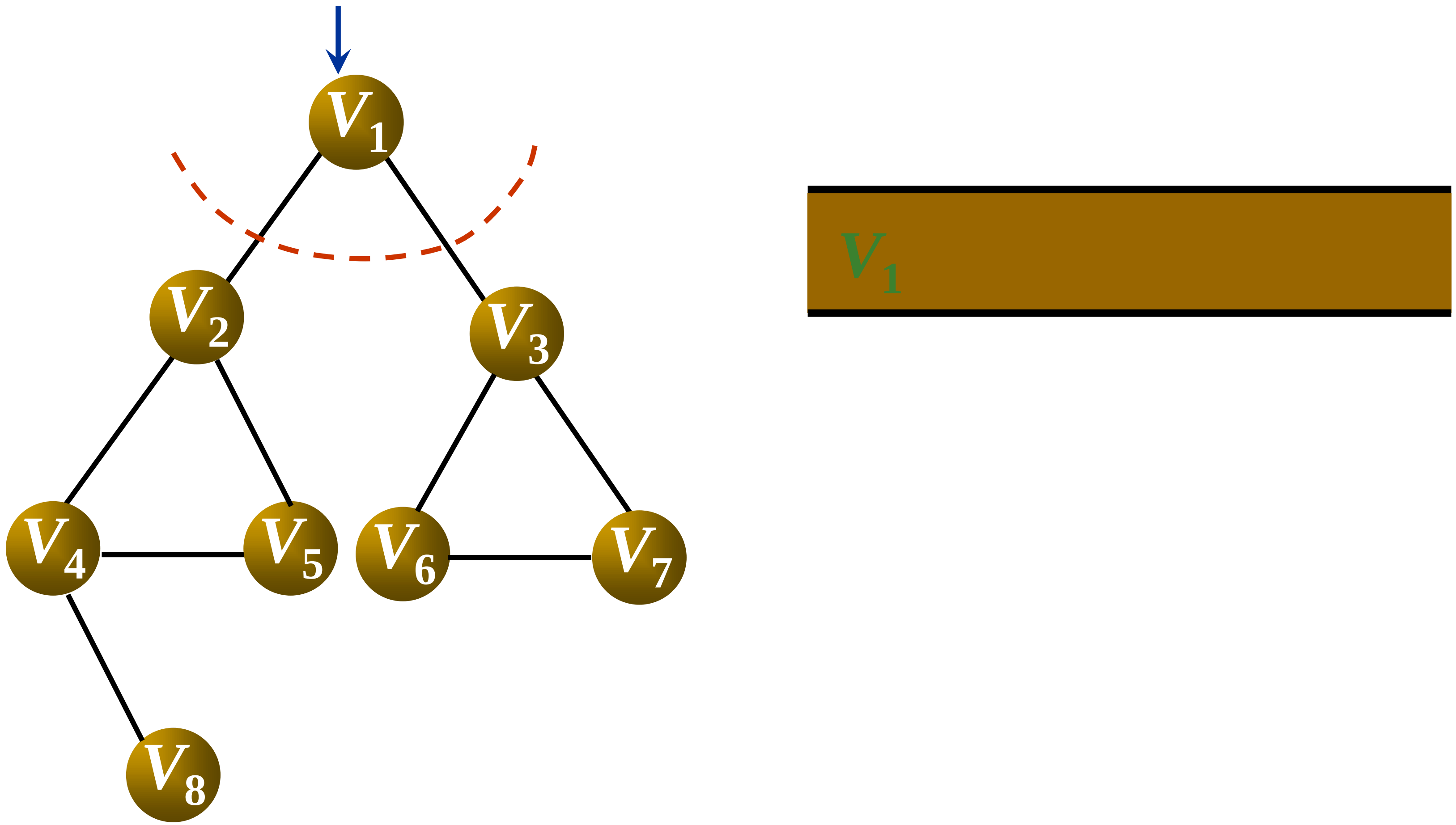
- <u>BFS and DFS differ only in the order in which they explore vertices.</u>

Breadth-First Search (BFS)

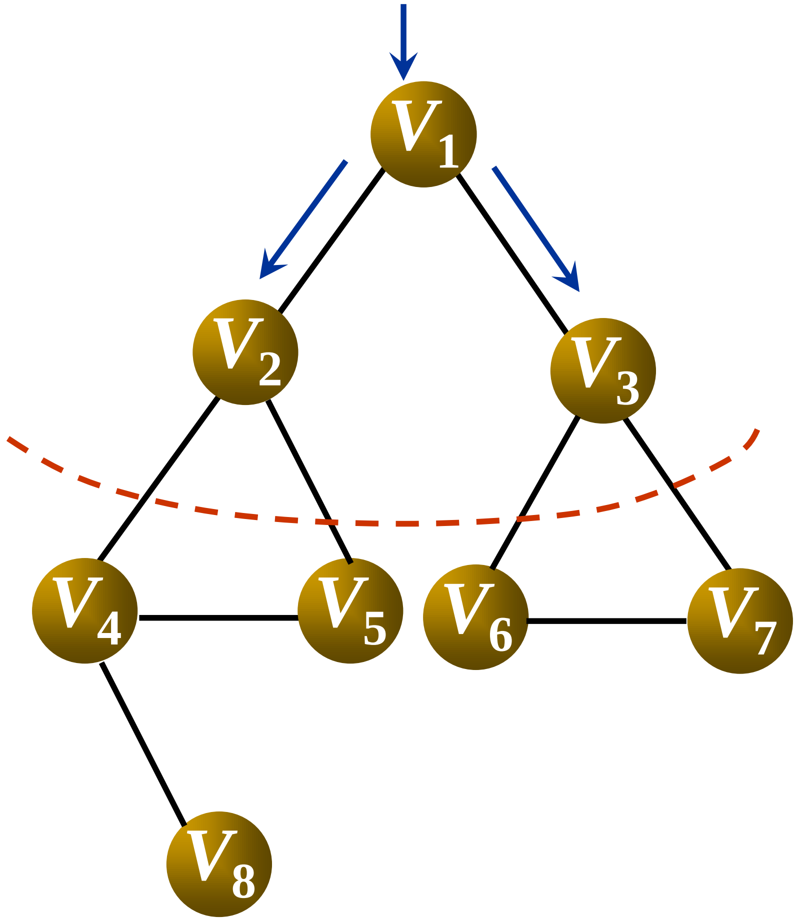
- A Breadth-First Search (BFS) traverses a connected component of a graph, and in doing so defines a spanning tree with several useful properties.
- The starting vertex *s* has level 0, and, as in DFS, defines that point as an “anchor.”
- In the first round, the string is unrolled the length of one edge, and all of the edges that are only one edge away from the anchor are visited.
- These edges are placed into level 1
- In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and placed in level 2.
- This continues until every vertex has been assigned a level.
- The label of any vertex ***v*** corresponds to the length of the shortest path from ***s*** to ***v***.

BFS - A Graphical Representation

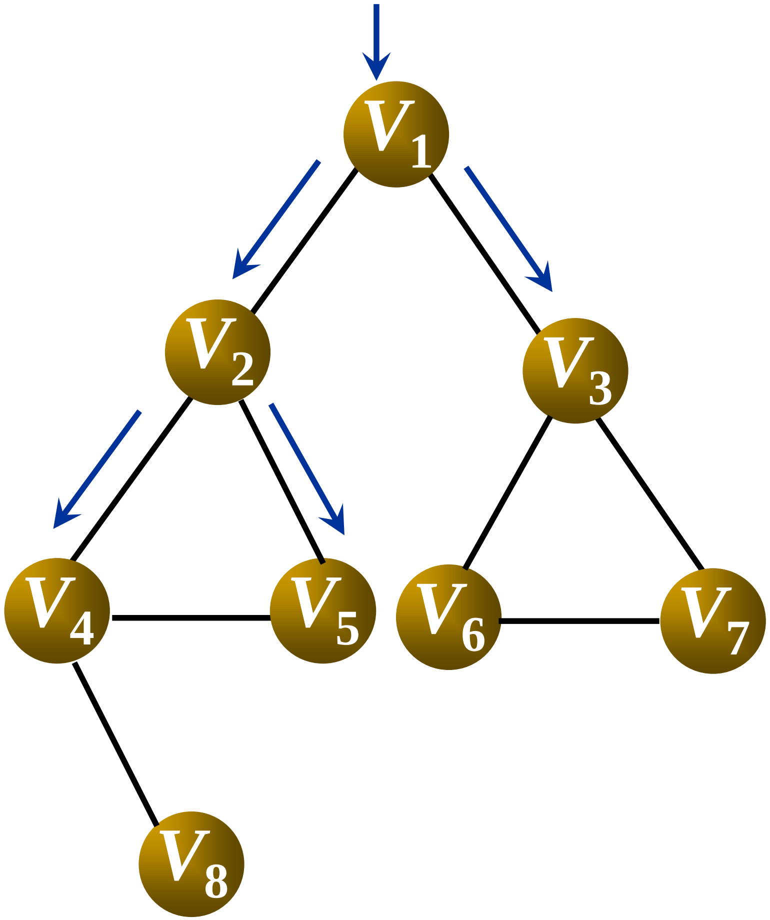




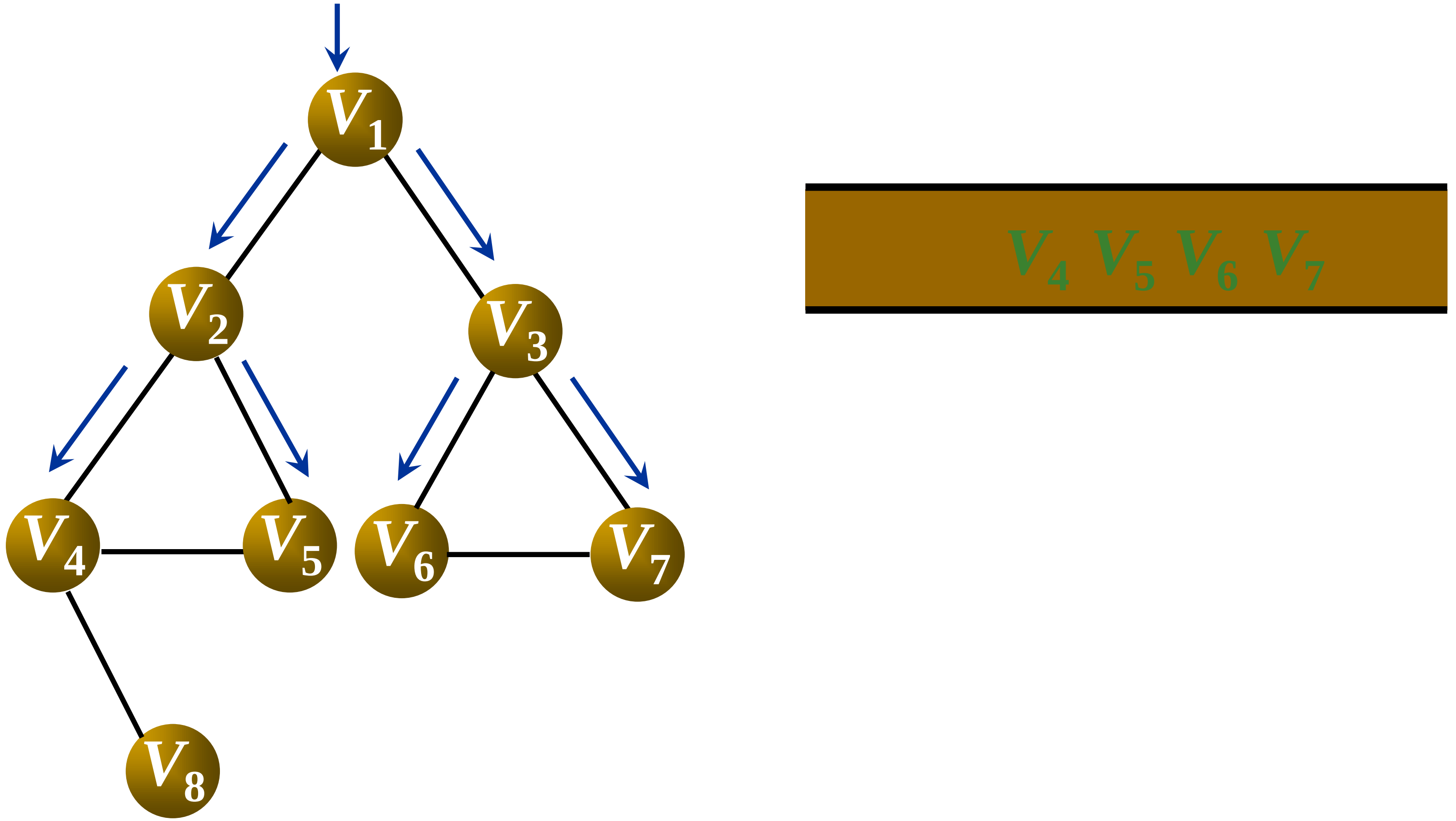
Result: V_1



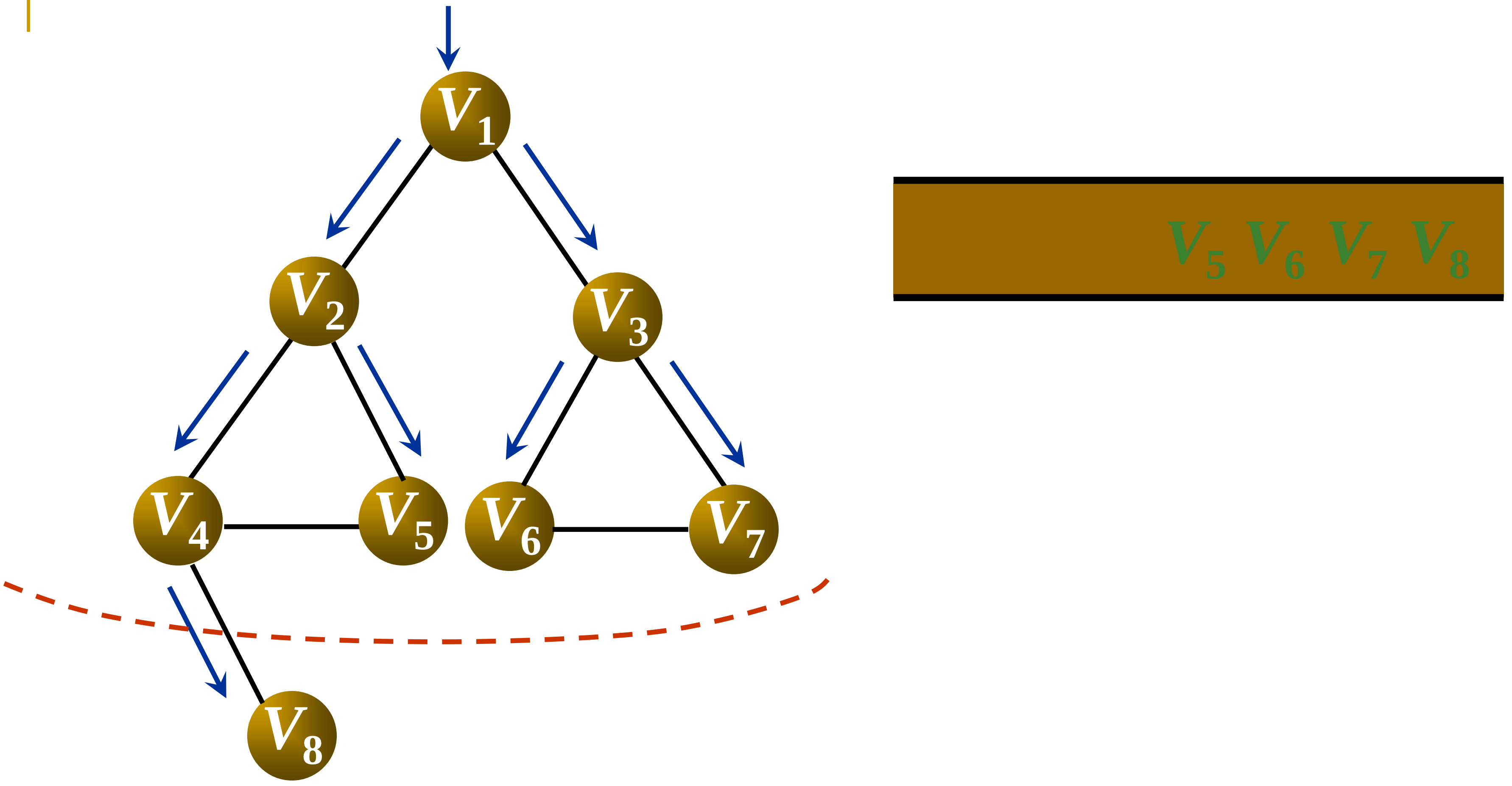
Result: V_1 V_2 V_3



Result: V_1 V_2 V_3 V_4 V_5



Result: V_1 V_2 V_3 V_4 V_5 V_6 V_7



Result: V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8

Code for BFS

```
void BFS( TABLE T )
{
    QUEUE Q;
    vertex v, w;
    Q = create_queue( NUM_VERTEX ); make_null( Q );
    enqueue( s, Q );
    while( !is empty( Q ) )
    {
        v = dequeue( Q );
        T[v].known = TRUE; /* not really needed anymore */
        for each w adjacent to v
            if( T[w].dist = INT_MAX )
            {
                T[w].dist = T[v].dist + 1;
                T[w].path = v;
                enqueue( w, Q );
            }
    }
dispose_queue( Q );
```

Code – BFS Using Adjacent Matrix

```
template <class T>
void MGraph::BFSTraverse(int v)
{
    front=rear=−1;   //Assuming no overflow
    cout<<vertex[v]; visited[v]=1;  Q[++rear]=v;
    while (front!=rear)
     {
        v=Q[++front];
        for (j=0; j<vertexNum; j++)
           if (arc[v][j]==1 && visited[j]==0 ) {
                 cout<<vertex[j]; visited[j]=1; Q[++rear]=j;
           }
      }
}
```

Code – BFS Using Adjacent List

```
template <class T>
void ALGraph::BFSTraverse(int v)
{
   front=rear=-1;
   cout<<adjlist[v].vertex;    visited[v]=1;   Q[++rear]=v;
   while (front!=rear)
   {
      v=Q[++front];    p=adjlist[v].firstedge;
      while (p!=NULL)
      {
         j= p->adjvex;
         if (visited[j]==0) {
            cout<<adjlist[j].vertex;  visited[j]=1; Q[++rear]=j;
         }
         p=p->next;
      }
   }
}
```

BFS Usage

- Finding a path

  In a maze, how to find a path from the start to the end using least steps?

- Exploiting traversal