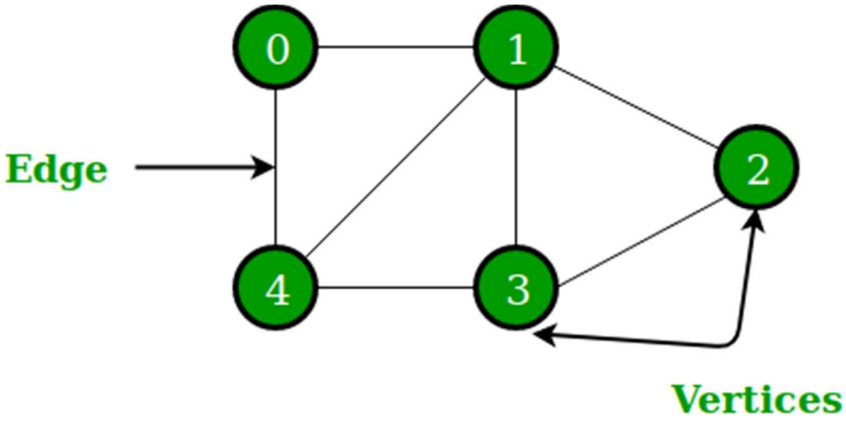
Graph: Data Structure and Algorithms

- A Graph is a non-linear data structure consisting of nodes and edges.
- The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
- More formally a Graph can be defined as,

A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.



In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

- Graphs are used to solve many real-life problems.
- Graphs are used to represent networks.
- The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook.
- For example, in Facebook, each person is represented with a vertex (or node).
- Each node is a structure and contains information like person id, name, gender, locale etc.

Graph and its representations

A graph is a data structure that consists of the following two components:

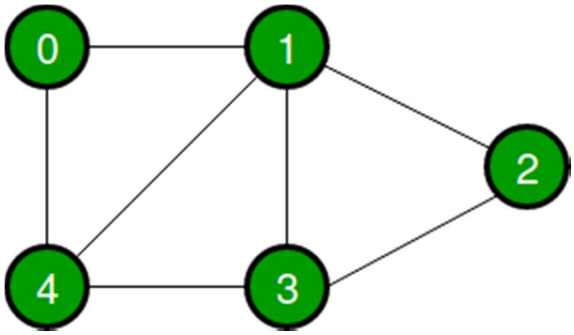
1. A finite set of vertices also called as nodes.

2. A finite set of ordered pair of the form (u, v) called as edge.

- The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph (di-graph).

- The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.
- Graphs are used to represent many real-life applications:
- Graphs are used to represent networks.

Following is an example of an undirected graph with 5 vertices.



The following two are the most commonly used representations of a graph.

1. Adjacency Matrix

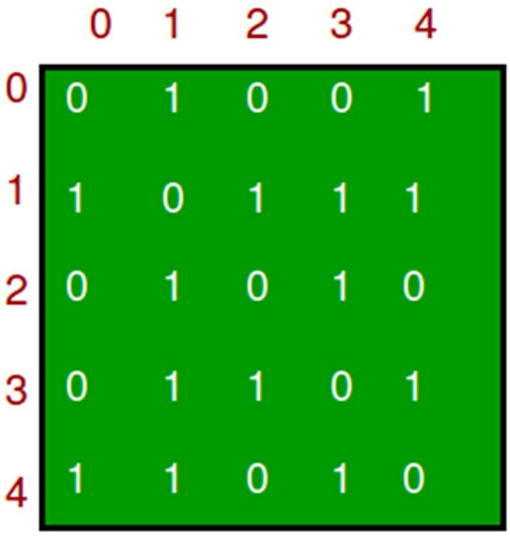
2. Adjacency List

- There are other representations also like, Incidence Matrix and Incidence List.
- The choice of graph representation is situation-specific.
- It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs.
- If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Pros:

- Representation is easier to implement and follow.
- Removing an edge takes O(1) time.
- Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

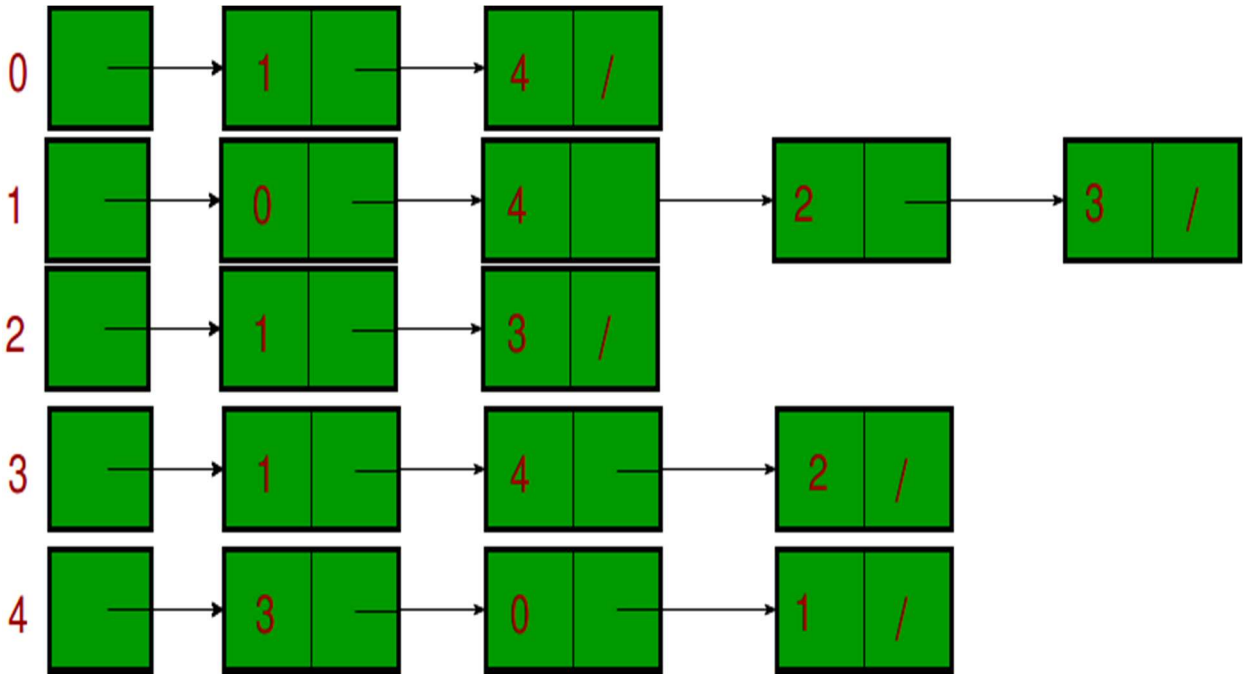
Cons:

- Consumes more space O (V^2).
- Even if the graph is sparse (contains less number of edges).
- It consumes the same space. Adding a vertex is O (V^2) time.
-

Adjacency List:

- An array of lists is used.
- The size of the array is equal to the number of vertices.

- Let the array be an array[].
- An entry array[i] represents the list of vertices adjacent to the *i*th vertex.
- This representation can also be used to represent a weighted graph.
- The weights of edges can be represented as lists of pairs.
- Following is the adjacency list representation of the above graph.




```
// A simple representation of graph using STL
#include<bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

```
// A utility function to print the adjacency list
// representation of graph
void printGraph(vector<int> adj[], int V)
{
    for (int v = 0; v < V; ++v)
    {
        cout << "\n Adjacency list of vertex "
             << v << "\n head ";
        for (auto x : adj[v])
           cout << "-> " << x;
        printf("\n");
    }
}

// Driver code
int main()
{
    int V = 5;
    vector<int> adj[V];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    printGraph(adj, V);
```

```
    return 0;
}
```

Output:

```
Adjacency list of vertex 0
head -> 1-> 4

Adjacency list of vertex 1
head -> 0-> 2-> 3-> 4

Adjacency list of vertex 2
head -> 1-> 3

Adjacency list of vertex 3
head -> 1-> 2-> 4

Adjacency list of vertex 4
head -> 0-> 1-> 3
```

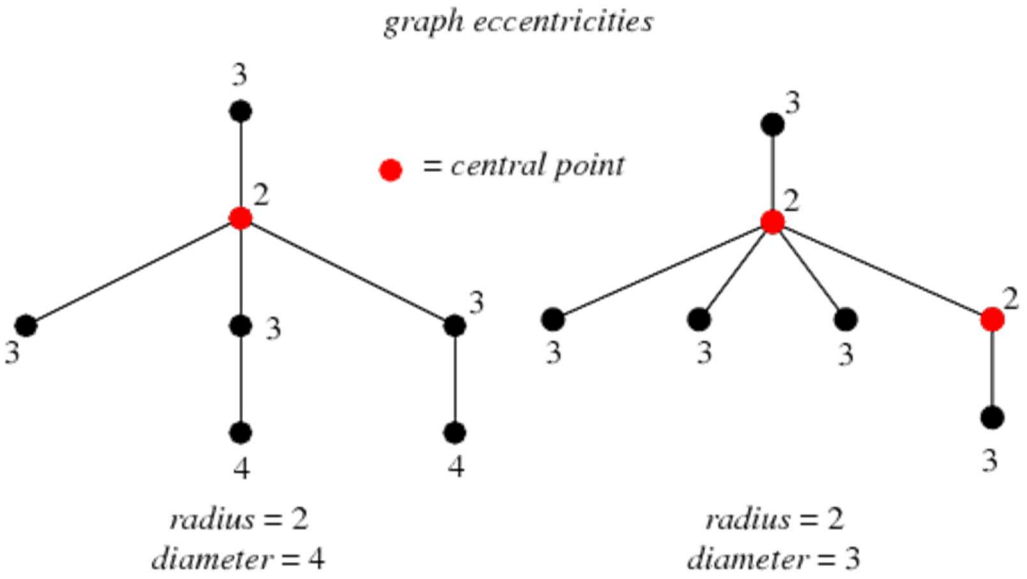
Pros:

- Saves space O(|V|+|E|) .
- In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.

Cons:

- Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

Graph Eccentricity



- The eccentricity $\epsilon(v)$ of a graph vertex v in a connected graph G is the maximum graph distance between v and any other vertex u of G.
- For a disconnected graph, all vertices are defined to have infinite eccentricity.
- The maximum eccentricity is the graph diameter.
- The minimum graph eccentricity is called the graph radius.
- Eccentricities are implemented as Eccentricity[*g*] .
-
- A nonstandard version of graph eccentricity for a given vertex v is implemented as VertexEccentricity[*g*, *v*], which gives the eccentricity for the connected component in which v is contained.
- Precomputed standard eccentricities (assuming infinite values for disconnected graphs) for a number of named graphs can be obtained using GraphData[*graph*, "Eccentricities".

Shortest-Paths Problems

Shortest-Paths Problems

- On a road map, a road connecting two towns is typically labeled with its distance.
- We can model a road network as a directed graph whose edges are labeled with real numbers.
- These numbers represent the distance (or other cost metric, such as travel time) between two vertices.
- These labels may be called *weights*, *costs*, or *distances*, depending on the application.
- Given such a graph, a typical problem is to find the total length of the shortest path between two specified vertices.

- This is not a trivial problem, because the shortest path may not be along the edge (if any) connecting two vertices, but rather may be along a path involving one or more intermediate vertices.
- **For example, in Figure the cost of the path from *A*** to *B* to *D* is 15.
- The cost of the edge directly from *A* to *D* is 20. The cost of the path from *A* to *C* to *B* to *D* is 10.
- Thus, the shortest path from *A* to *D* is 10 (rather than along the edge connecting *A* to *D*). We use the notation **d**(*A*,*D*)=10 to indicate that the shortest distance from *A* to *D* is 10.
- In Figure there is no path from *E* to *B*, so we set **d**(*E*,*B*)=∞.
- We define **w**(*A*,*D*)=20 to be the weight of edge (*A*,*D*), that is, the weight of the direct connection from *A* to *D*.
- Because there is no edge from *E* to *B*, **w**(*E*,*B*)=∞. Note that **w**(*D*,*A*)=∞ because the graph of Figure is directed. We assume that all weights are positive.

Single-Source Shortest Paths

- We will now present an algorithm to solve the *single-source shortest paths problem*. Given Vertex *S* in Graph **G**, find a shortest path from *S* to every other vertex in **G**.
- We might want only the shortest path between two vertices, *S* and *T*.
- However in the worst case, finding the shortest path from *S* to *T* requires us to find the shortest paths from *S* to every other vertex as well.
- So there is no better algorithm (in the worst case) for finding the shortest path to a single vertex than to find shortest paths to all vertices.
- The algorithm described here will only compute the distance to every such vertex, rather than recording the actual path.
- Recording the path requires only simple modifications to the algorithm.
- Computer networks provide an application for the single-source shortest-paths problem.
- The goal is to find the cheapest way for one computer to broadcast a message to all other computers on the network.
- The network can be modeled by a graph with edge weights indicating time or cost to send a message to a neighboring computer.
- For unweighted graphs (or whenever all edges have the same cost), the single-source shortest paths can be found using a simple breadth-first search.
- When weights are added, BFS will not give the correct answer.

One approach to solving this problem when the edges have differing weights might be to process the vertices in a fixed order.

- Label the vertices *v*0 to *vn*−1, with *S*=*v*0.
- When processing Vertex *v*1, we take the edge connecting *v*0 and *v*1.
- When processing *v*2, we consider the shortest distance from *v*0 to *v*2 and compare that to the shortest distance from *v*0 to *v*1 to *v*2.

- When processing Vertex *vi*, we consider the shortest path for Vertices *v*0 through *vi*−1 that have already been processed.
- Unfortunately, the true shortest path to *vi* might go through Vertex *vj* for *j*>*i*. Such a path will not be considered by this algorithm.
- However, the problem would not occur if we process the vertices in order of distance from *S*.
- Assume that we have processed in order of distance from *S* to the first *i*−1 vertices that are closest to *S*; call this set of vertices **S**.
- We are now about to process the *i* th closest vertex; call it *X*

.

- A shortest path from *S* to *X* must have its next-to-last vertex in *S*

. Thus,

$\mathbf{d}(S,X)=\min_{U\in\mathbf{S}}(\mathbf{d}(S,U)+\mathbf{w}(U,X))$.

- In other words, the shortest path from *S* to *X* is the minimum over all paths that go from *S* to *U*, then have an edge from *U* to *X*, where *U* is some vertex in **S**.
- This solution is usually referred to as Dijkstra's algorithm. It works by maintaining a distance estimate **D**(*X*) for all vertices *X* in **V**.
- The elements of **D** are initialized to the value INFINITE. Vertices are processed in order of distance from *S*. Whenever, a vertex *v* is processed, **D**(*X*) is updated for every neighbor *X* of *V*.
- Here is an implementation for Dijkstra's algorithm.
- At the end, array D will contain the shortest distance values.

```
// Compute shortest path distances from s, store them in D
static void Dijkstra(Graph G, int s, int[] D) {
  for (int i=0; i<G.nodeCount(); i++)    // Initialize
    D[i] = INFINITY;
  D[s] = 0;
  for (int i=0; i<G.nodeCount(); i++) {  // Process the vertices
    int v = minVertex(G, D);     // Find next-closest vertex
    G.setValue(v, VISITED);
    if (D[v] == INFINITY) return; // Unreachable
    int[] nList = G.neighbors(v);
    for (int j=0; j<nList.length; j++) {
      int w = nList[j];
      if (D[w] > (D[v] + G.weight(v, w)))
        D[w] = D[v] + G.weight(v, w);
    }
  }
```