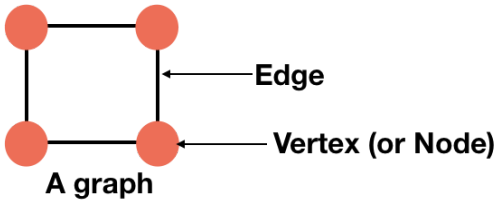
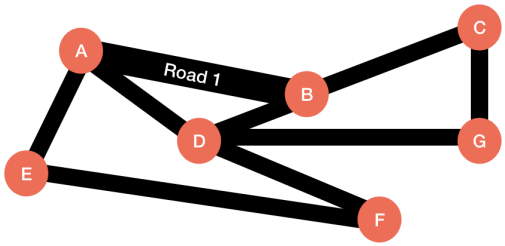
Graphs

In simplest terms, a graph is a combination of vertices (or nodes) and edges.



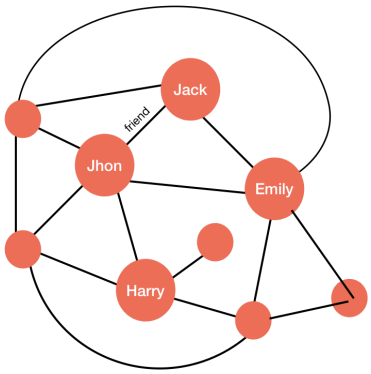
In the above picture, we have 4 nodes and 4 edges and it is a graph.

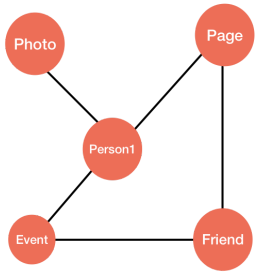
Graph is a very important data structure to store data which are connected to each other. The simplest example is the network of roads to connect different cities.



We can see how different cities and roads are mapped into different nodes and edges to form a graph.

There are many other relationships which are stored using graphs efficiently. For example, electrical circuits, people network on social media like Facebook, etc.





We represent a graph as G=(V, E), where V is the set of all of its vertices and E is the set of all of its edges.

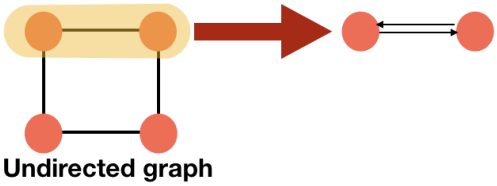
We are going to use |V| and |E| to represent the number of vertices and number of edges respectively.

Let's have a look at different types of graphs which are generally used.

Types of Graphs

Undirected Graph and Directed Graph

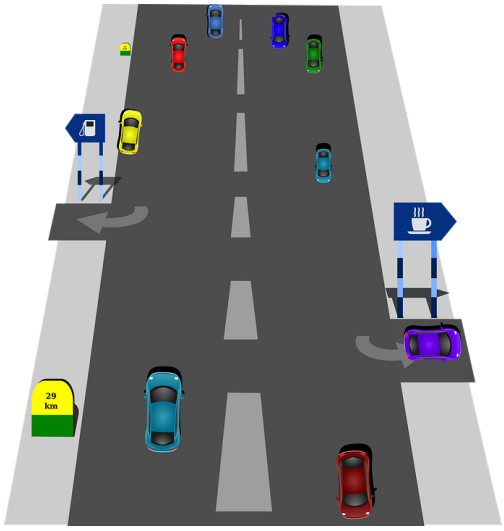
A graph in which if there is an edge connecting two vertices A and B, implies that B is also connected back to A is an **undirected graph**.



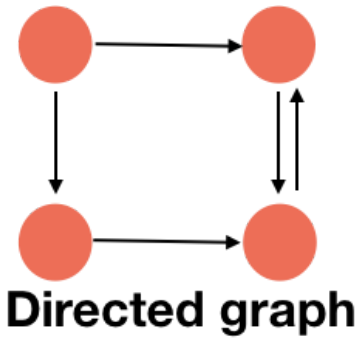
In the above graph, 1 is connected to 2 and 2 is connected back to 1 and this is true for every edge of the graph. So, the graph is an undirected graph.

In other words, we can say that a graph G=(V,E) is undirected if an edge (i, j) (edge from i to j) ∈ E implies that the edge (j, i) ∈ E also.

Two lane roads on which vehicles can go in either direction are the example of undirected graphs.



In a **directed graph**, the edges are directed i.e., they have a direction. So, if the edge is an edge from i to j i.e., (i, j) ∈ E, then this doesn't necessarily mean that there will be also an edge from j to i.

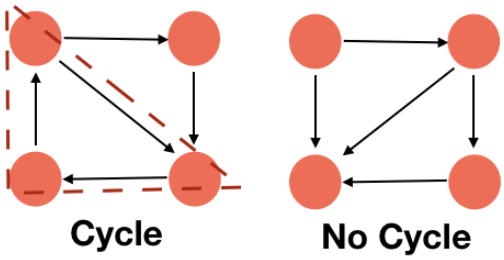


This is a directed graph as there is a path from 1 to 2 but there isn't any path from 2 to 1.

Suppose a person is following someone on Twitter but may or may not be followed back. This can be represented by directed graphs.

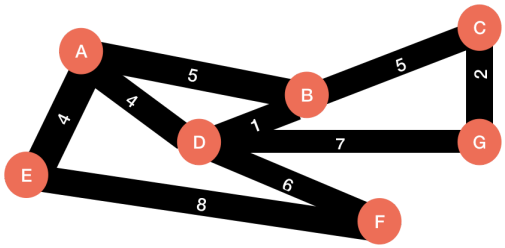
Cyclic and Acyclic Graph

In a cyclic graph, there are cycles or loops formed by the edges but this doesn't happen with an acyclic graph.



Weighted and Unweighted Graph

Sometimes weights are given to the edges of a graph and these are called weighted graphs. For example, in a graph representing roads and cities, giving the length of the road as weight is a logical choice.



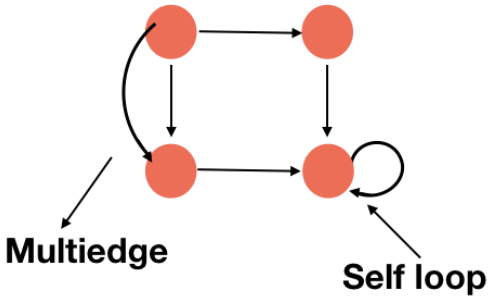
Dense and Sparse Graph

When the number of edges ($|E|$) is close to the square of the number of vertices ($|V|^2$), then the graph is a dense graph. To visualize, you can imagine a graph with few vertices and lots of edges between them.

If $|E|$ is much less than $|V|^2$, then it is a sparse graph.

Simple and Non-simple Graph

The graph which has self-loops or an edge (i, j) occurs more than once (also called multiedge and graph is called multigraph) is a non-simple graph.

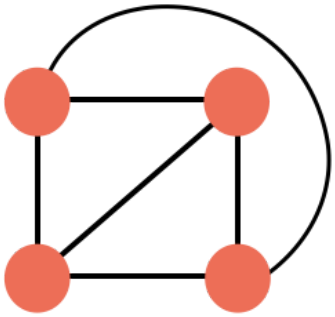


Connected and Disconnected Graph

If every node of a graph is connected to some other nodes is a connected graph. However, if there is at least one node which is not connected to any other node, then it is a disconnected graph.

Complete Graph

When each node of a graph is connected to every other node, then it is called a complete graph.



Complete graph

Now, we have an idea of what basically is a graph. Now, we need to know how to use a graph in our program and for that we need to learn how to represent a graph.

Representation of Graphs

There are two ways of representing a graph:

- Adjacency-list representation
- Adjacency-matrix representation

According to their names, we use lists in the case of adjacency-list representation and a matrix (2D array) in the case of adjacency matrix representation.

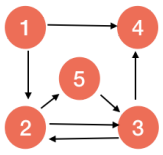
The way in which we are going to represent our graph depends on the task we have to perform. This will be clear after studying these representations in detail.

Adjacency-matrix Representation

In the adjacency-matrix representation, we use a $|V|x|V|$ matrix to represent a graph.

	1	2	3	4	5
1					
2					
3					
4					
5					

Here, we are assuming that the vertices of the graph are numbered from 1 to $|V|$. Now for the cell (i, j) of this matrix, we set its value 1 if we have an edge from i to j (or $(i, j) \in E$), otherwise 0.



	1	2	3	4	5
1	0	1	0	1	0
2	0	0	1	0	1
3	0	1	0	1	0
4	0	0	0	0	0
5	0	0	1	0	0

In the above picture, we have an edge from 1 to 2, so the cell (1, 2) is 1, but there is no edge connecting 2 back to 1, so the cell (2, 1) is 0.

Similarly, there is no edge from the vertex 4 to any other vertices, so (4, 1), (4, 2), ... , (4, 5) all are 0.

It is quite clear that we are using a |V|x|V| matrix to represent a graph G=(V, E), so the space required to store this would be $\Theta(|V|^2)$.

Now imagine a social media site like Facebook, where there are billions of users. Using a matrix will take quite a large amount of space but the real problem is that even among these billions of people, a person is not connected most of the other person i.e., if each person on average has 1000 friends, then most of the cells are going to be empty (or 0). In this case (and most of the cases), we use adjaceny-list representation.

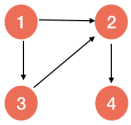
However, there is also one advantage of this representation - we can tell if one node is connected to another node in no time ($\Theta(1)$ to be appropriate).

We can say that using an adjacency-list for a sparse graph and adjacency-matrix for a dense graph is a general choice.

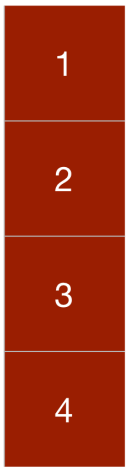
In most of the applications, the number of nodes which are connected from a node is much less than the total number of nodes. So, we move to adjacency-list representation.

Adjacency-list Representation

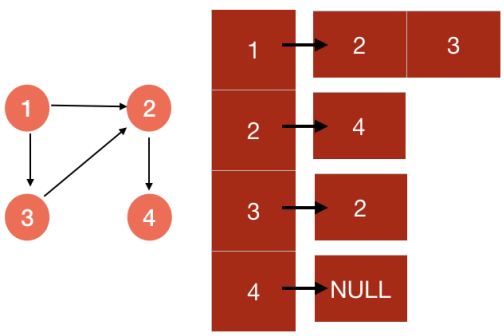
In adjacency-list representation, we have a list of all the nodes and for each node, we have a list of nodes for which the node has an edge.



For example, for the above graph, we will have a list of all the nodes.



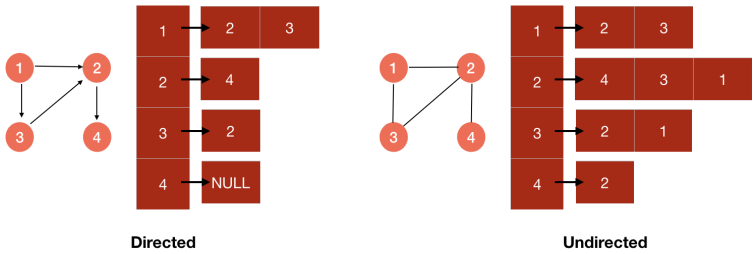
Now, for each of these nodes, we will store a list (array, hash or linked list) of other nodes which have an edge from these nodes (or adjacent nodes). So, node 1 will have a list containing nodes 2 and 3, node 2 will have a list containing node 4, etc.



Thus, in an adjency-list representation, we have an array (Adj) of |V| lists and each element of this array Adj i.e., Adj[i] consists all the vertices adjacent to it.

(/add_quest

The total number of items in all these lists is the number of edges in the graph i.e., |E|, if the graph is directed. But if the graph is undirected, then the total number of items in these adjacency lists will be 2|E| because for any edge (i, j), i will appear in adjacency list j and vice-versa.



Now, the total space taken to store this graph will be space needed to store all adjacency list + space needed to store the lists of vertices i.e., |V|.

So, it requires $\Theta(|V| + |E|)$ space.

We can also add a new vertex in an adjacency -list in $O(1)$ time but doing the same will require $O(|V|^2)$ time in the case of an adjacency-matrix representation because a |V+1|x|V+1| matrix is needed to be reconstructed.

You can learn about linked lists from C++ : Linked lists in C++ (Singly linked list) (https://www.codesdope.com/blog/article/c-linked-lists-in-c-singly-linked-list/) and Linked lists in C (Singly linked list) (https://www.codesdope.com/blog/article/linked-lists-in-c-singly-linked-list/) articles.

C **Python** **Java**

(/add_ques

```
#include <stdio.h>
#include <stdlib.h>

/*
  Node for linked list of adjacent elements.
  This will contain a pointer for next node.
  It will not contain the real element but the index of
  element of the array containing all the vertices V.
*/
typedef struct list_node {
  int index_of_item;
  struct list_node *next;
}list_node;

/*
  Node to store the real element.
  Contain data and pointer to the
  first element (head) of the adjacency list.
*/
typedef struct node {
  int data;
  list_node *head;
}node;

/*
  Graph will contain number of vertices and
  an array containing all the nodes (V).
*/
typedef struct graph{
  int number_of_vertices;
  node heads[]; // array of nodes to store the list of first nodes of each adjacency list
}graph;

/*
array heads       Adjacency list. contain index and next
contain data
and head of
adjacency list
 ____             ____    ____    ____
|node|    ----> |list|->|    |->|    |
|____|           |node|   |____|  |____|
 ____             ____    ____    ____
|    |    ----> |    |->|    |->|    |
|____|           |____|   |____|  |____|
 ____             ____    ____    ____
|    |    ----> |    |->|    |->|    |
|____|           |____|   |____|  |____|
 ____             ____    ____    ____
|    |    ----> |    |->|    |->|    |
|____|           |____|   |____|  |____|

*/

node* new_node(int data) {
  node *z;
  z = malloc(sizeof(node));
  z->data = data;
  z->head = NULL;

  return z;
}

list_node* new_list_node(int item_index) {
  list_node *z;
  z = malloc(sizeof(list_node));
  z->index_of_item = item_index;
  z->next = NULL;

  return z;
}

// make a new graph
graph* new_graph(int number_of_vertices) {
  //number_of_vertices*sizeof(node) is the size of the array heads
  graph *g = malloc(sizeof(graph) + (number_of_vertices*sizeof(node)));
  g->number_of_vertices = number_of_vertices;
```

(/add_quest

```
  //making elements of all head null i.e.,
  //their data -1 and next null
  int i;
  for(i=0; i<number_of_vertices; i++) {
    node *z = new_node(-1); //*z is pointer of node. z stores address of node
    g->heads[i] = *z; //*z is the value at the address z
  }

  return g;
}

// function to add new node to graph
void add_node_to_graph(graph *g, int data) {
  // creating a new node;
  node *z = new_node(data);
  //this node will be added into the heads array of the graph g
  int i;
  for(i=0; i<g->number_of_vertices; i++) {
    // we will add node when the data in the node is -1
    if (g->heads[i].data < 0) {
      g->heads[i] = *z; //*z is the value at the address z
      break; //node is added
    }
  }
}

// function to add edge
void add_edge(graph *g, int source, int dest) {
  int i,j;
  // iterating over heads array to find the source node
  for(i=0; i<g->number_of_vertices; i++) {
    if(g->heads[i].data == source) { //source node found

      int dest_index; //index of destination element in array heads
      // iterating over heads array to find node containg destination element
      for(j=0; j<g->number_of_vertices; j++) {
        if(g->heads[j].data == dest) { //destination found
          dest_index = j;
          break;
        }
      }

      list_node *n = new_list_node(dest_index); // new adjacency list node with destination index
      if (g->heads[i].head == NULL) { // no head, first element in adjaceny list
        g->heads[i].head = n;
      }
      else { // there is head which is pointer by the node in the head array
        list_node *temp;
        temp = g->heads[i].head;

        // iterating over adjaceny list to insert new list_node at last
        while(temp->next != NULL) {
          temp = temp->next;
        }
        temp->next = n;
      }
      break;
    }
  }

  // if the graph is undirected, there will also be edge from dest to source
  for(i=0; i<g->number_of_vertices; i++) {
    if(g->heads[i].data == dest) {

      int source_index;

      for(j=0; j<g->number_of_vertices; j++) {
        if(g->heads[j].data == source) {
          source_index = j;
          break;
        }
      }

      list_node *n = new_list_node(source_index);
      if (g->heads[i].head == NULL) {
        g->heads[i].head = n;
      }
      else {
```

(/add_quest

```
        list_node *temp;
        temp = g->heads[i].head;

        while(temp->next != NULL) {
          temp = temp->next;
        }
        temp->next = n;
      }
      break;
    }
  }
}

void print_graph(graph *g) {
  int i;
  for(i=0; i<g->number_of_vertices; i++) {
    list_node *temp;
    temp = g->heads[i].head;
    printf("%d\t",g->heads[i].data);
    while(temp != NULL) {
      printf("%d\t",g->heads[temp->index_of_item].data);
      temp = temp->next;
    }
    printf("\n");
  }
}

int main() {
  graph *g = new_graph(4);
  add_node_to_graph(g, 1);
  add_node_to_graph(g, 2);
  add_node_to_graph(g, 3);
  add_node_to_graph(g, 4);
  add_edge(g, 1, 2);
  add_edge(g, 1, 3);
  add_edge(g, 2, 4);
  add_edge(g, 3, 2);
  print_graph(g);
  return 0;
}
```

Now, we know what is a graph and how to use it. So, let's move ahead and study some algorithms related to the graphs.

> “ Life is a bad teacher. First, it puts a test, and then teaches. ”

PREV (/course/algorithms-huffman-codes/) (/course/algorithms-bfs/) NEXT

Further Readings

- Linked lists in C (Singly linked list) (/blog/article/linked-lists-in-c-singly-linked-list/)
- C++ : Linked lists in C++ (Singly linked list) (/blog/article/c-linked-lists-in-c-singly-linked-list/)

Download Our App.