Graph Algorithms

Recommended graph book for programmers

By Greeshma Amaraneni

CHAPTERS

1. Introduction and Basics
    - Introduction to Graphs
    - Types of Graphs
    - Graph Representation
    - Applications of Graphs

2. Graph Traversals
    - Introduction
    - Depth First Traversal
    - Breadth First Traversal
    - Applications
    - Problems

3. Cycles
    - Introduction
    - Cycle Detection in Directed Graph
    - Cycle Detection in Undirected Graph
    - Negative Weight Cycle
    - Applications
    - Problems

4. Tree algorithms and Queries
    - Introduction
    - Disjoint Set
    - Tree Queries
    - Problems

5. Minimum Spanning Tree
    - Introduction
    - Kruskal's Algorithm
    - Prim's Algorithm
    - Applications
    - Problems

6. Shortest paths
    - Introduction
    - Dijkstra Algorithm
    - Bellman Ford Algorithm
    - Floyd Warshall Algorithm
    - Applications
    - Problems

7. Connectivity
    - Introduction
    - Kosaraju Algorithm
    - Applications
    - Problems

8. Flows and Cuts
    - Introduction
    - Ford Fulkerson Algorithm
    - Relation between Max Flows and Min Cut
    - Applications
    - Problems

CHAPTER 1

INTRODUCTION AND BASICS

Section 1.1- Introduction to Graphs

Many practical problems can be solved by modeling the given problem as a graph problem and applying a suitable graph algorithm.The book mainly concentrates on the fundamental graph algorithms required to solve the basic programming problems.In this chapter we will be introduced to the graph terminology and the representation of graphs.

What is a Graph?

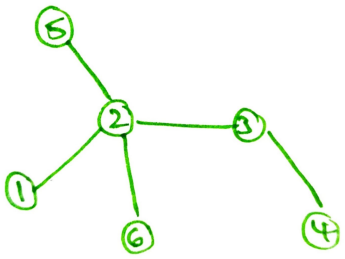
A Graph is a non-linear data structure that consists of a set of nodes and edges connecting two nodes.In simple words it is an ordered pair G=(V,E) where V is collection of nodes and E is collection of edges.

Example:

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

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

Corresponding Graph G=(V,E) is



Graph Terminology:

We will be discussing the general graph terminologies which denotes the parameters of the graph that will be used in this book.

Node:

Node is simply the fundamental unit of a graph.It is also called a vertex.

Edge:

Edge represents a path between two nodes or simply a line between two nodes.

Path:
A path in a graph is the sequence of edges which joins a sequence of nodes. The length of the path is the same as the number of edges present in the path.

Cycle:
A cycle in a graph is a path that starts and ends at the same node.

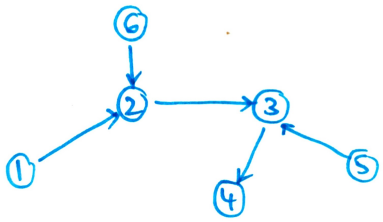
Degree:
The Degree of a vertex in a graph is the number of edges incident on it.For a directed graph the outdegree of a vertex is the total number of outgoing edges and similarly the indegree of a vertex is the total number of incoming edges.

Tree:
Tree is a connected graph that has no cycles.

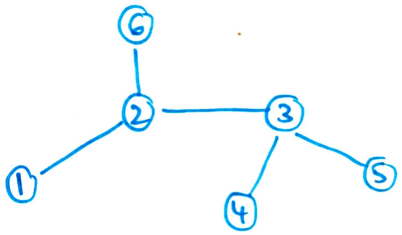
Section 1.2 - Types of Graphs

Directed and Undirected Graph:

Directed Graph:
A directed graph is a set of nodes or vertices connected together where all the edges are directed from one node to the other.
Example:



Undirected Graph:
An undirected graph is a set of nodes where the direction of connection between two nodes is non-important i.e all the edges are bidirectional.
Example:

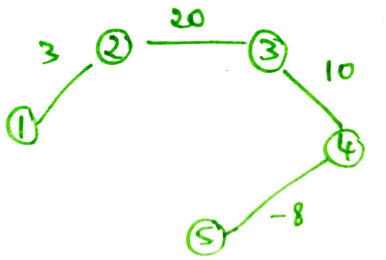


Weighted and Unweighted Graph:

Weighted Graph:

Weighted Graph is a graph that has some numerical weight on all the edges of a graph.

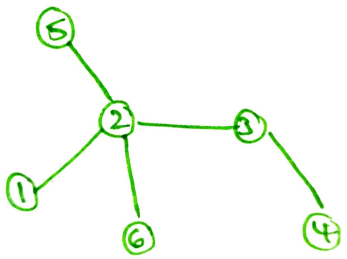
Example:



Unweighted Graph:

Unweighted Graph is a graph that has no weight on the edges in the graph.

Example:

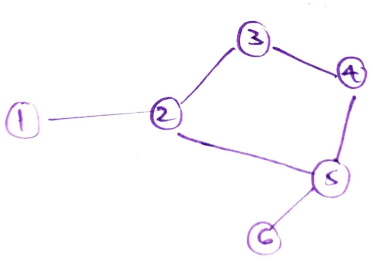


Simple and Multi Graph:

Simple Graph:

Simple graph is a graph that is undirected,unweighted and it doesn't contain any multi edges or self loops.

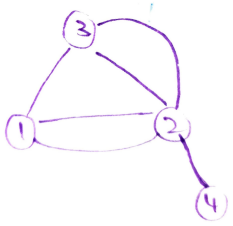
Example:



Multli Graph:

Multi graph is an undirected graph in which multiple edges are allowed(and sometimes self loops) i.e two or more edges that connect the same two nodes.

Example:

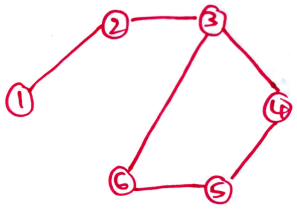


Cyclic and Acyclic Graph:

Cyclic Graph:

A graph contains at least one cycle i.e there exists at least one path such that starting from a node we can reach the same node by traversing through edges.

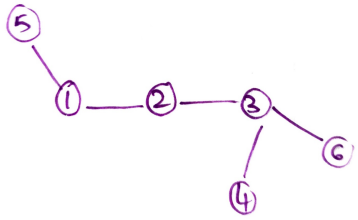
Example:



Acyclic Graph:

A graph that doesn't contain any cycle is called an acyclic graph.Tree is also an acyclic graph and it comes under a special category of acyclic graph where nodes present in the tree are connected.

Example:

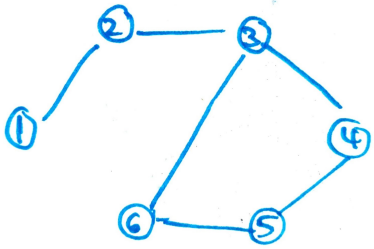


Connected and Disconnected Graph:

Connected Graph:

Connected Graph is a graph that has a path between a pair of vertices i.e there are no unreachable vertices.

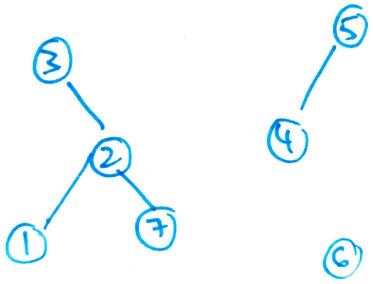
Example:



Disconnected Graph:

The graph which is not connected is called a disconnected graph.

Example:

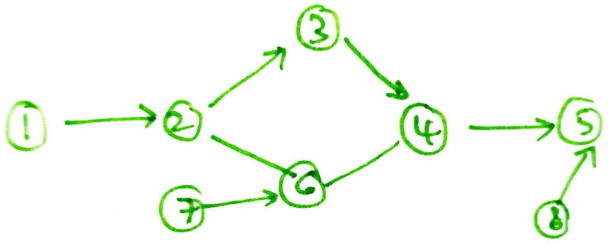


Special Graphs:

Directed Acyclic Graph:

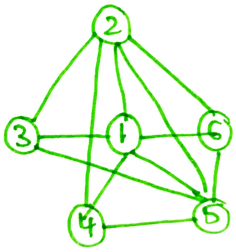
Directed Acyclic graph is a directed graph that has no cycles.

Example:



Sparse Graph:
Sparse graph is a graph where the number of edges present in the graph is much less than the number of edges possible in a graph.
Example:

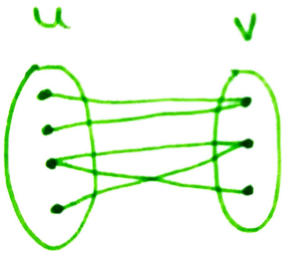
Dense Graph:
Dense graph is a graph where the number of edges present in the graph is close to the number of edges possible in the graph.
Example:



Bipartite Graph:

Bipartite graph is a graph whose nodes can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V.

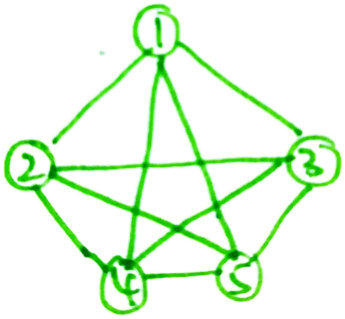
Example:



Complete Graph:

Complete graph is a graph having an edge between every two nodes. It has no loops and every two nodes share exactly one edge.

Example:



Section 1.3- Graph Representation

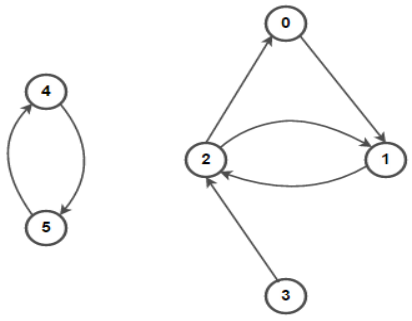
However the choice of graph representation is situation dependent,the common ways of graph representation is adjacency matrix ,adjacency list and edge list.Apart from these three there are some other representations like edge list representation,incident matrix ,incidence list and so on.This book mainly concentrates on adjacency list,adjacency matrix and edge list representation.We shall also compare the adjacency list and matrix and discuss which is preferred when as well.

Adjacency Matrix:

Adjacency matrix representation consists of a 2D array of size n*n where n is the number of vertices.Let us suppose the adjacency matrix to be adj[n][n] where

For a non-weighted graph adj[i][j] = 1 implies there is an edge between node i and node j and for a weighted graph adj[i][j]=0 implies there is no edge between nodes i and j.

Let us consider below graph,



The corresponding adjacency matrix is

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

Advantages:

- Implementation is easy
- Finding the existence of edge between two nodes is easy i.e it takes just O(1) time

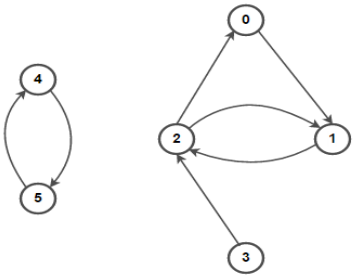
Disadvantages:

- The matrix consumes too much of space i.e O(n*n)
- Adding or deleting a vertex takes O(n*n)
- It is slow to iterate over all edges.

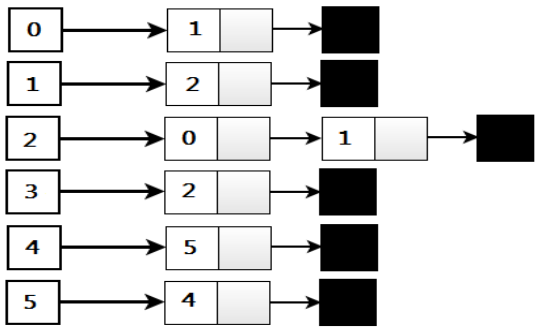
Adjacency list:

Adjacency list representation consists of an array of lists.The number of array elements is the same as the number of nodes and each element is a list that consists of the elements that are adjacent to this node.

Let us consider below graph,



The corresponding adjacency list is



Advantages:

- It consumes less amount of space i.e space complexity is of the order O(|v|+|e|)
- Adding a vertex is easy i.e it just consumes O(1) time.
- Removing a vertex is also easy compared to adjacency matrix i.e it just consumes O(|v|+|e|) time.
- Fast to iterate over all edges

Disadvantages:

- Finding the existence of edge between two nodes takes more time relatively i.e O(|V|) in the worst case.

Adjacency Matrix Vs Adjacency List:

We usually get puzzled which graph representation would be better between the adjacency matrix and adjacency list for solving a given problem.Select the right representation for graph based on below aspects:

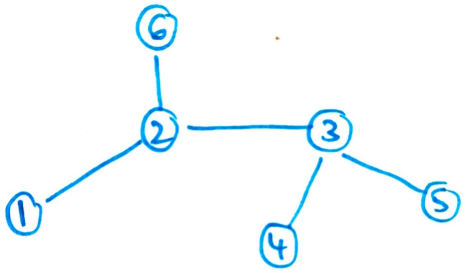
1. *Sparse/Dense*:
   If the graph is dense then an adjacency matrix would be preferred, else an adjacency list would be a good choice as it just stores the required adjacent nodes.

2. *Iteration over edges:*
   If the problem involves frequently iterating over all edges of some node or all edges then the adjacency list would work well as it keeps the list of adjacent nodes.
3. *Lookup:*
   If some query requires checking the presence of edge between two nodes or the weight of an edge then adjacency matrix would be preferred as it consumes less time.

Edge List:

Edge List representation of graph consists of the set of edges along with corresponding nodes in an array.For weighted graph it stores the weight of the edge as well.The edge list representation is used in algorithms that need to iterate through all edges and process queries in the algorithm like Prim's Algorithm,Kruskal Algorithm,Dijkstra,Bellman Ford and many such algorithms.

Let us consider the below graph,



The corresponding edge list representation for the above graph is [(1,2),(2,6),(2,3),(3,4),(3,5)]

This book illustrates many examples where some problems are solved using an adjacency matrix,some are using an adjacency list and others using edge list representations which would help in better understanding what to implement in some given situation.

Section 1.4- Applications of Graphs

Graphs are widely used in the real world scenarios which are used to define the flow of computation and also represent networks of communication and data organization as well.The algorithms involved in Graphs are used to solve the problems of representing graphs as networks and the way internet is connected and also exploit social network connectivity like the social media apps.Since the graph algorithms have wide applications in the real life it is important to learn how the algorithm is designed and drawbacks of algorithms and relate the graph algorithms discussed to the daily life scenarios.

The common applications of graphs are:

- *Google Maps:*
  It uses graphs for building transportation systems where the roads are considered as edges and the intersection of roads as vertices and hence the navigation system uses the Graph algorithms to find the shortest paths between the two vertices like Dijkstra,Bellman Ford,Floyd Warshall algorithms and many other algorithms.

- *Facebook:*
  The users of facebook are considered to be the vertices and an edge between any two vertices represent that they are friends and friends suggestion depends on an algorithm used that finds the friends of friends and other social media apps also use the graph theory.

- *World Wide Web:*
  The webpages are considered as vertices and a link from one page to another page represents that there is a directed edge from one vertex to other and this could be modelled well using Graphs.

CHAPTER 2

GRAPH TRAVERSAL

Section 2.1 - Introduction

In this chapter the two main graph algorithms involved are depth first traversal and breadth first traversal that are used to traverse the graph when a starting node is provided. The algorithms vary in the order they visit the adjacent nodes each time.Let us look at the formal definition of these traversals.

Depth First Traversal:
Depth First Traversal or DFS is another graph traversal where from a starting node,we keep on traversing branches of each node until the leaf node is reached and then we backtrack the nodes until all the nodes of a given starting or root node are processed.

Breadth First Traversal:
Breadth First Traversal or BFS is another graph traversal where we explore all nodes at current depth level before going to next depth levels from a starting node.Distance from a starting node to all other nodes can be calculated using BFS i.e starting from the root or starting node it explores nodes present at depth 1 and next depth 2 and so on.

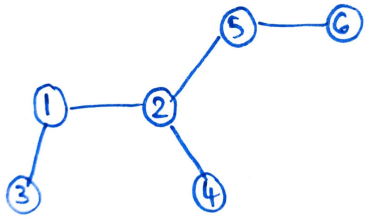
Note:
All nodes will be reached by considering any starting node in a connected graph using any Graph traversal.

In the next sections we will be discussing the algorithm for DFS and BFS along with an example for each and look at their Code implementations and complexity analysis as well.Also the applications of DFS and BFS algorithms will be discussed and some exercise problems related to them are present at the end of the chapter.

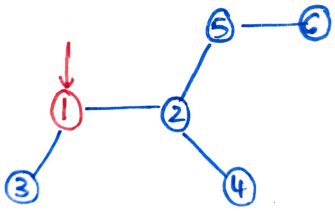
Section 2.2 - Depth First Traversal

In this section we will be looking at algorithms for DFS traversal with an example and the code implementation using two methods: Recursive calls and Queue Data structure.

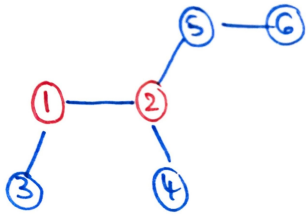
Example:
Let us understand DFS traversal using the below undirected graph:



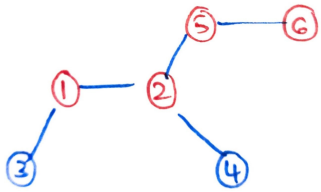
Considering the starting node as 1 let us apply DFS on node 1 and get the corresponding graph traversal.



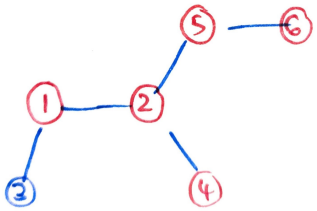
Let us first consider node numbered as 2 for processing it's children i.e DFS on node 2 and after backtracking from node we process node 3.



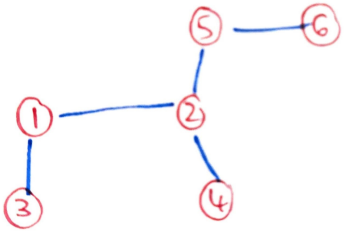
Node 2 has two branches and among 4 and 5,let us traverse node 5 first and after processing it we shall come to node 4.



Node 5 has only one child i.e node 6 so the next node in the traversal is node 6 and then we backtrack to node 2 since node 5 has got only 1 child and from node 2 we go to it's next child i.e node 4.



So far we have processed all children of node 2 so now upon backtracking we are left with node 3 which is the child of root or starting node 1.



Node 3 hasn't got any children so it's the end of graph traversal,so the order we obtained is,

1,2,5,6,4,3

DFS traversal can be implemented using two methods:

Implementation using Stack:

1. **Algorithm:**
    - We keep track of each node if it is already present in the stack or not.
    - First the given node will be pushed into the stack and the one of the adjacent nodes that is not present in the stack will be pushed into the stack.
    - If no such node is present then we backtrack popping out that node and now we check for the adjacent nodes of the top element and accordingly we either push it's children or pop that element out.
    - Each time we push some element into a stack we either print it or store them in a separate array to print the order of traversal at the end.

2. **Pseudo code:**

```
while(!IsemptyStack())//checking if stack is empty
node=top()
flag=0
for(child in node)//checking all branches of it are visited or not
      If visited(child)==0
       push(child)//pushing the child into stack
          flag=1 //if it's not present in stack
          break
      }
      If flag==0
      pop()//popping the node out if all are processed
```

3. **Code:**

   DFS_stack.cpp

4. **Analysis:**

   Time Complexity: O(n+e)

   The code is implemented using adjacency list representation and all the nodes will be pushed to stack exactly once we loop through all edges in the worst case.So time complexity is O(n+e) where n is number of nodes and e is number of edges.

   Space Complexity: O(1)

   In the code implementation extra space is not consumed in any functions.So space complexity is constant i.e O(1).

Implementation using recursive function:

1. **Algorithm:**
   - We keep track of each node if it is already explored or not.
   - A recursive function will be called by starting node initially where it loops through all adjacent nodes of the calling node.
   - If any node is not explored then a recursive call will be made by that node else we return from the function.
   - After each recursive call made,we print the calling node or store it and the order of recursive calls is required graph traversal.
2. **Pseudo code:**

```
DFS(node):
   Is_visited(node)=1;
   print(node); //either print each time or store all
   for child in adjancent_nodes[node]
       if (Is_visited(child)==0)
           DFS(child);
   return;
```

3. **Code:**

   DFS_recursion.cpp

4. **Analysis:**

   Time Complexity: O(n+e)

   The code is implemented using adjacency list representation and the DFS calls are recursively called for n nodes and at max e edges will be reached.So time complexity is O(n+e) where n is number of nodes and e is number of edges.

   Space Complexity: O(1)

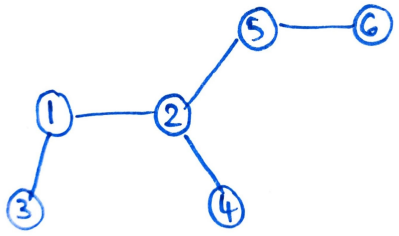
   In the code implementation extra space is not consumed in any functions or recursive calls.So space complexity is constant i.e O(1).

Section 2.3 - Breadth First Traversal

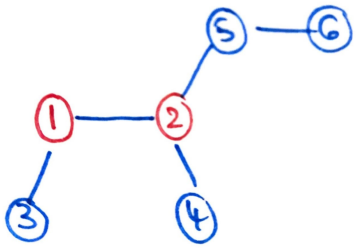
In this section we will be looking at algorithms for BFS traversal with an example and the code implementation using the Stack data structure and look at pseudo code as well.

Example:

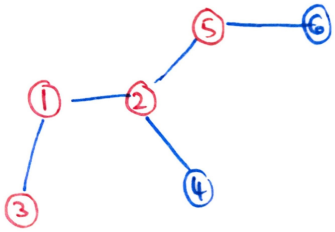
Let us understand BFS traversal using the below undirected graph:



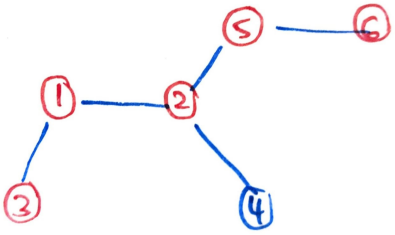
Considering the starting node as 1 let us apply BFS on node 1 and get the corresponding graph traversal.



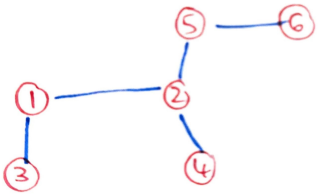
The nodes present at depth 1 for the node 1 are 2 and 3 so the graph traversal starting from 1 is followed by 2 and 3.



Now we need to get the nodes at depth 1 corresponding to nodes 2 and 3 ,so 4 and 5 being the nodes with a depth 1 from node 2 will be considered in the next level traversal.



In the next level node 6 is present at depth 3 from starting node 1 as it's depth 1 from node 5 so this is present in the next exploration of the graph.



This is the end of graph traversing since all nodes have been explored and hence the order of BFS traversal is 1,2,3,4,5,6

Implementation using Queue:

1. **Algorithm:**
   - Push the root node into the queue.
   - If the queue is not empty then pop the top element and print or store it as well and push it's adjacent nodes into the Queue if they are not explored so far.
   - Repeat the above step till the Queue becomes empty.
   - The order of popping out of elements from Queue is the required Breadth First Traversal.

2. **Pseudo Code:**

```
push(start_node)
while(!Is_Queue_Empty)
      int elem=Top()
      pop()
      print(elem)//print or store popped out element
      for it in adjacent_nodes[elem]
          if it=not_visited
              push(it)
              it=visited
```

3. **Code:**

   BFS.cpp

4. **Analysis:**

   Time Complexity: O(n+e)

   The code is implemented using adjacency list representation and the DFS calls are recursively called in the functions with maintaining three sets.So time complexity is O(n+e) where n is number of nodes and e is number of edges.

   Space Complexity: O(1)

   In the code implementation extra space is not consumed in any functions or recursive calls.So space complexity is constant i.e O(1).

Section 2.4 - Applications

The graph traversing algorithms BFS and DFS are widely used in the other graph algorithms as well.Some of the applications of these graph traversals has been discussed below.

- *Cycle detection in graph*: DFS or BFS can be used to detect cycles in a directed or undirected graph with slight modifications in their implementation as a back edge implies a cycle detected in the graph.

- *Path finding:* For finding a path between two vertices either BFS or DFS traversals can be used by simply modifying them to return from the functions if the destination node is reached from a given source node.

- *Checking if the graph is bipartite:* Using colors along with either BFS or DFS by simply coloring two adjacent nodes with different colors and finding the presence of an edge between two colors in the traversal and deciding accordingly.

- *Strongly connected components in graph:* For finding the strongly connected components in a graph the algorithms like Kosaraju and Tarjan which would be discussed in the coming chapters of the book uses DFS search.

- *Topological Sorting:* In a Directed Acyclic graph inorder to find the topological order of nodes DFS is used in the algorithm and it is discussed in coming chapters of the book.

- *Ford-Fulkerson Algo*: For finding the maximum flow in a graph from a source vertex to a destination vertex,BFS search is used to find the path between these vertices as the shortest path is needed in the Ford-Fulkerson algorithm which is clearly explained in the Flows and Cuts chapter.

- ## Section 2.5- Problems

Some good problems taken from various coding platforms based on graph algorithms discussed in this chapter are given below.Give them a try!!

1. ShortestReach
2. Tour
3. Even Outdegree Edges
4. Graph
5. Labyrinth
6. Monsters

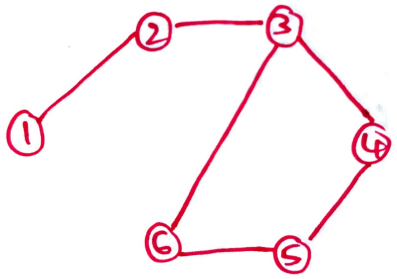
CHAPTER 3

CYCLES

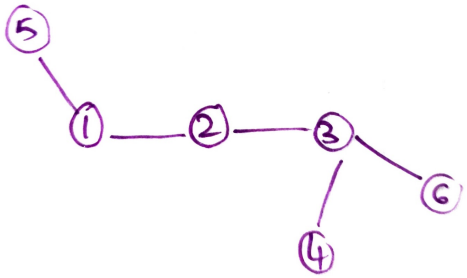
Section 3.1 - Introduction

Cycle:
A path starting at some node and after traversing through edges and ending up on the same node is called a cycle.

Cyclic Graph:
A graph containing at least one cycle i.e there exists at least one path such that starting from a node we can reach the same node by traversing through edges.
Example:



Acyclic Graph:
A graph that doesn't contain any cycle is called an acyclic graph.Tree is also an acyclic graph and it comes under a special category of acyclic graph where nodes present in the tree are connected.
Example:

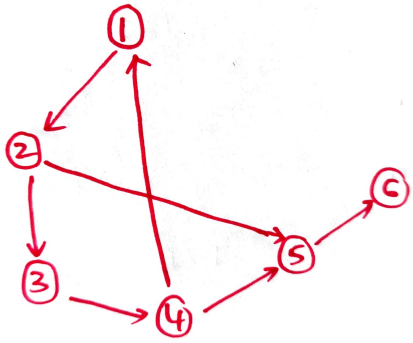


Detecting cycles in a graph is one of the frequent queries needed in many problems and algorithms.For detecting a cycle in directed and undirected graphs,the methods graph coloring and DFS algorithms are used respectively which are discussed in next sections.Also we will be having a small discussion on negative edge cycle and look at some problems based on cycles at the end of this chapter.

Section 3.2 - Cycle detection in Directed Graph

In this section we will be looking at an algorithm used for detecting a cycle in an directed graph using DFS with graph coloring and also look at an example solved using the algorithm and code implementation of the algorithm followed by the complexity analysis.

Consider the below directed graph,



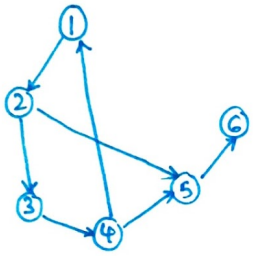
The above graph is cyclic as we can see a cycle starting at node 1 and again reaching 1 through directed path 1-2-3-4-1.We can detect the presence of a cycle in a directed graph using the method of graph coloring using the algorithm discussed below.

Algorithm:

1. We maintain three sets of nodes,the first set contains all the nodes that are not visited and the second set has all the nodes which are visited and in the stage of processing and the last set contains all the nodes that are explored.
2. Initially all are not visited and we apply DFS on all nodes that are not yet visited through looping over all of them and we recursively apply DFS calls on a node and their children and in this stage they go to set-2 as they are in still progress and once DFS on a node is completely over i.e as that node is completely explored we send that node to last set.
3. In the DFS call of some node if we see it's child belonging to the second set then it's clear that there is some other path from that node to it's child other than the direct edge between them which indicates that there is a cycle in the graph.

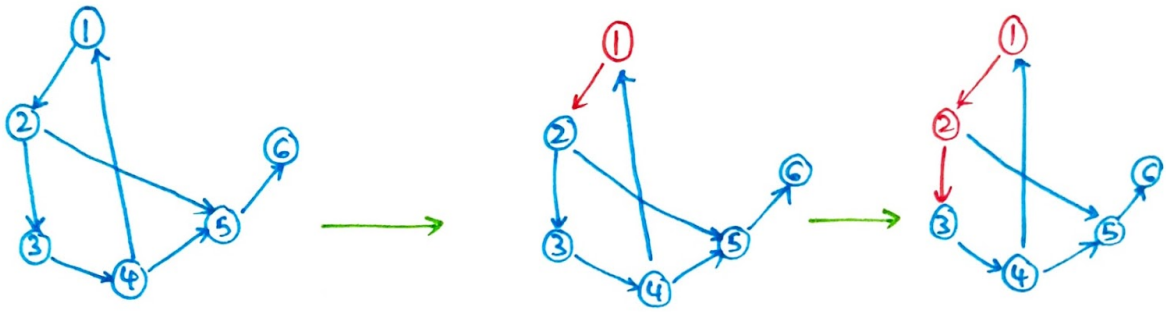
Example:

Consider the below directed graph,

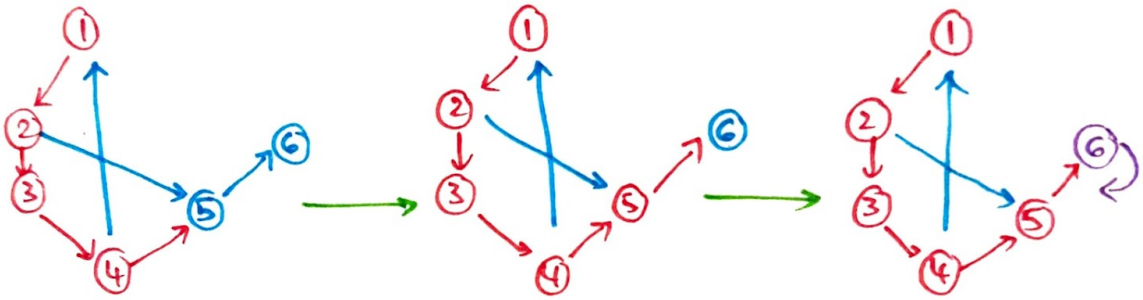


Nodes with blue color belong to the first set which are not visited,nodes with red color belong to the second set which are still in the stage of processing and the nodes in violet color are explored completely.

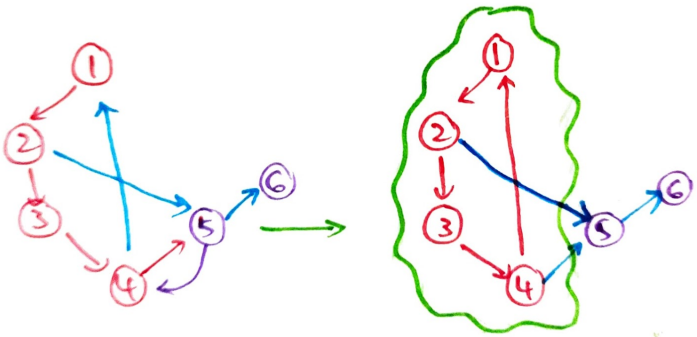
Starting the DFS traversal at node 1 we explore node 2 and among the adjacent nodes of 2 as 3 and 5 we first DFS on node 3 and continue traversing till node 3 and 4.



We have 2 adjacent nodes i.e 1 and 5 at node 4 and let us choose node 5 to traverse next and in the next stage node 6 is traversed and since node 6 doesn't have no children we backtrack from it and pushing it to the set of explored nodes and similarly we backtrack to node 4 from node 5.



Now we explored one child of node 4 and we need to go for other node i.e node 1 but we observe that node 1 and 4 are already in the set of processing nodes,so according to our algorithm as we found an edge between two processed nodes a cycle is present in the graph.



As seen in the above diagram there is a cycle in the graph with nodes 1,2,3,4.

Code Implementation:

Pseudo code:

```
int DFS(start_node):
  set[start_node]=2
  for child in start_node:
    if set[child]==2
      return 1
    if set[child]==1
       DFS(child)
 return 0


for node in All_nodes[]:
  if(DFS(node)==1)
  ans=true
  return
ans=false
return
```

Code:

DirectedCycle.cpp

Analysis:

Time Complexity: O(n+e)

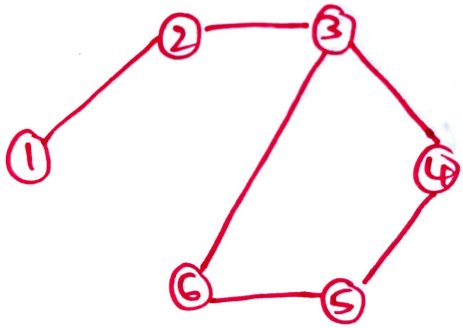
The code is implemented using adjacency list representation and the DFS calls are recursively called in the functions with maintaining three sets.So time complexity is O(n+e) where n is number of nodes and e is number of edges.

Space Complexity: O(1)
In the code implementation extra space is not consumed in any functions or recursive calls.So space complexity is constant i.e O(1).

Section 3.3 - Cycle detection in Undirected Graph

In this section we will be looking at an algorithm used for detecting a cycle in an undirected graph using DFS.We also look at an example solved using the algorithm and code implementation of the algorithm followed by the complexity analysis.

For example consider the below undirected graph,



The above graph is cyclic as we can see the path 3-4-5-6-3 which is forming a cycle in the graph.We can detect the presence of a cycle in an undirected graph using simple DFS using the algorithm discussed below.

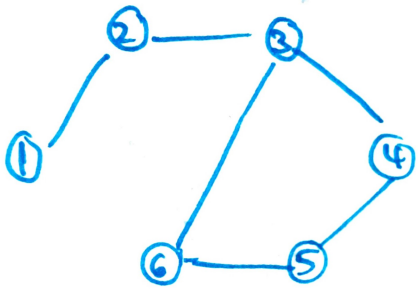
Algorithm:

1. In this approach we keep track of a node if it is visited in DFS or not and unlike in procedure of direct graph we don't maintain three sets of nodes.
2. Initially all are not visited and we apply DFS on all nodes that are not yet visited through looping over all of them and we recursively apply DFS calls on a node and their children and their state is changed from unvisited to visited.

3. In the DFS call while iterating through each child of a node,if the status of some child is visited and if that is not parent of the node then definitely there is path from node to child other than the edge between them indicating the presence of cycle in undirected graph.

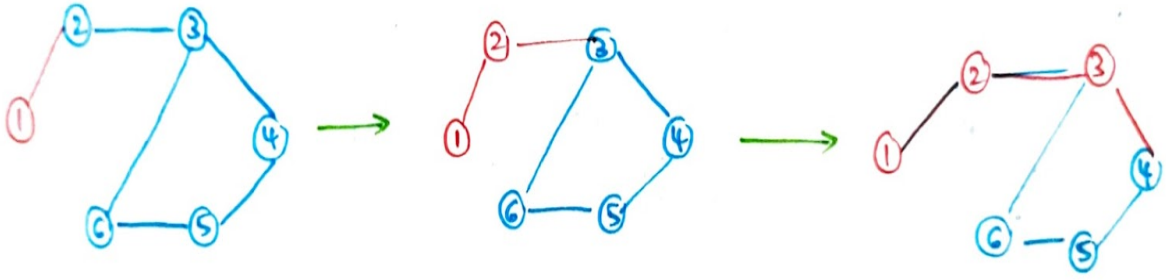
Example:

Consider the below undirected graph,

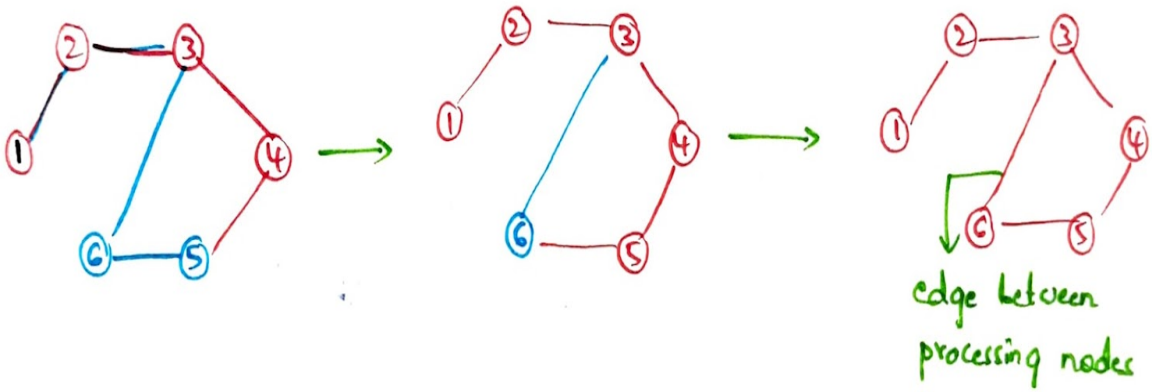


In the diagrams given below the nodes with blue color indicate the nodes that are not visited yet and the nodes in red color are visited.So let us solve the above problem using the algorithm discussed.

Starting at node 1 we traverse node 2 and 3 in the DFS traversal and put the nodes 1,2,3 in visited set of nodes and now 3 has two adjacent nodes 3 and 4 and let us choose node 4 in the DFS traversal and then process node 6.



From node 4 we got to node 5 and then node 6 in DFS traversal since node 4,5 has only two options as adjacent nodes and also we can't go back to the parent node so now we are at node 6 and we find that there is and edge between node 3 and node 6 i.e between two nodes which are in stage of processing and according to the algorithm there should exist a cycle in the graph.



Using the algorithm we detected a cycle in the graph containing the nodes 3,4,5,6.In the next section we will be looking at the code implementation of the above algorithm.

Code Implementation:

1. **Pseudo code:**

```
int DFS(int start_node,int parent):
 visited(start_node)=1
 for child in start_node[]:

    if visited(child)==0
       DFS(child,start_node)
    else if visited(child)==1 and child!=parent
       return 1
return 0
```

```
for node in All_nodes[]:
 if(DFS(node,All_nodes)==1)
 ans=true
 return
ans=false
return
```

2. **Code:**
   Undirected_cycle.cpp

3. **Analysis:**
   Time Complexity: O(n+e)
   The code is implemented using adjacency list representation and the DFS calls are recursively called in the functions with maintaining three sets.So time complexity is O(n+e) where n is number of nodes and e is number of edges.

   Space Complexity: O(1)
   In the code implementation extra space is not consumed in any functions or recursive calls.So space complexity is constant i.e O(1).

Section 3.4 - Negative weight cycle

Negative weight Cycle:
A Negative Edge cycle is a cycle having the sum of weights in the cycle as negative.

In an undirected graph if there is an edge with negative weight then we say that the graph has a negative edge cycle since an undirected edge is equivalent to two directed edges but in opposite direction so the nodes corresponding to the edge will form a cycle with sum of weights as negative.

Negative weight cycle detection:

1. A simple approach would be to keep calculating the sum of weights of edges of all cycles in a graph and accordingly decide the presence of a negative weight cycle in the graph and finding a cycle can be done using modified DFS traversals as explained earlier.
2. In Bellman Ford algorithm for calculating the shortest path,(discussed in chapter 6) relax all the edges the number of times equal to number of nodes and if there is any node that is relaxed in the last relaxation then there exists at least one negative weight edge cycle in the graph.
3. In Floyd Warshall algorithm as well for finding the shortest path from a source node,find the distance from node to itself and if that is found out to be negative then there exists a negative weight cycle in the graph.

Section 3.5 - Applications

Many algorithms require cycle detecting and proceeding further in their implementation including Prim's algorithm,Kosaraju algorithm and many other algorithms.Also some practical problems involve detection of cycles and refer to the next section of problems to get an idea in various fields cycle detection is used.

Section 3.6- Problems

Some good problems taken from various coding platforms based on graph algorithms discussed in this chapter are given below.Give them a try!!

1. Tree modification
2. Find out
3. What?
4. Messages
5. Entry
6. Weird Watch

CHAPTER 4

TREE ALGORITHMS & QUERIES

Section 4.1 - Introduction

In this chapter basic concepts of trees will be covered and then we shall look at some of the tree algorithms and then discuss disjoint set data structure which is used in graph theory for solving many problems and also see topological sorting.In this section we shall look at some basic terminology used in Trees.

Tree:
Tree is an undirected graph in which any two vertices are connected by exactly one path, or equivalently a connected acyclic undirected graph.

Leaves:
A leaf of an unrooted tree is a node of vertex degree 1

Root Node:
The root node is the highest node in the tree structure, and has no parent.

Spanning Tree:
A spanning tree is a subgraph of a given graph G that covers all nodes with a minimum number of edges.

Forest:
The collection of trees in graph is called as a forest

Tree Edge:
It is an edge which is present in the tree obtained after applying DFS on the graph.

Forward Edge:
It is an edge (u, v) such that v is descendant but not part of the DFS tree.

Back edge:
It is an edge (u, v) such that v is the ancestor of node u but not part of the DFS tree.

Directed Acyclic Graph:
Directed Acyclic Graph which is also called DAG is a directed graph with no directed cycles. That is, it consists of vertices and edges, with each edge directed from one vertex to another, such that following those directions will never form a closed loop.

Section 4.2 - Disjoint Set

In this section we shall look at a data structure disjoint set and also look at the operations of a disjoint set like union and find and look at the code implementation as well.

Disjoint set:
Disjoint set is a data structure used to keep track of a set of elements that are partitioned into sets which are not overlapping.In this book the disjoint set data structure is implemented using an array but usually an array or a linked list and the resulting one after the operations on a disjoint set would form a tree.

In the context of disjoint set we define the following terms for code implementation,
Parent:
Parent is a node that has an edge to a child node. Leaf is a node that does not have a child node in the tree.
Absolute parent:
It is the main ancestor of the tree or more appropriately it is considered as the root of a tree and the parent of the absolute parent is considered as none.

Operations in Disjoint Set:

1. **Find:**
   The operation enables us to know which subset a given element belongs to.This operation is more used to determine if two elements belong to the same set or different.
   - We find the absolute parent of each element and check if they are equal inorder to determine if both belong to the same set or not.

2. **Union:**
   The union of two elements belonging to two different sets is the same as the union of two sets.
   - For the union of two elements we find their absolute parents and set one of these absolute parents to the other so that all the elements in the set will be merged to a new set.

Union-Find Algorithm:

The Union-Find Algorithm is an algorithm which usually performs the two operations union and find.

Algorithm:

- Storing the parent of each element in an array consisting of all the nodes and in the initial stage there is no parent for each of the nodes.
- For find operation we find the absolute parent of some node by going to the nodes pointed by the current node until the final node has no parent.
- For union operation merging two nodes is the same as merging two sets so we will be finding the absolute parents for two of the nodes using the find operation and then we make one of the absolute parents point to the other absolute parent and finally get a single set.

Code Implementation:

1. **Pseudo code:**

```
//find operation
int find(x):
  while(parent(x)!=-1)
    x=parent(x)
  return x


//union operation
void Union(x,y):
 int Absparent1=find(x)
 int Absparent2=find(y)
 parent(Absparent1)=Absparent2

 return
```

2. **Code:**

   Union_Find.cpp

3. **Time Complexity:**

   In the worst case the tree has only one sided branches and in that case both the operations take O(n) to find the root and union them as well since the union operation also requires find operation.

Union by Rank and Path Compression:

In order to reduce the time complexity for the union and find operations we slightly modify the approach to improve the working of the Union-Find Algorithm.

Path Compression:

We observe that in naive find operation for calculating absolute parent for an element we need to take the same path of traversing the elements to reach the absolute parent each time and this could be improved by updating the parent of the elements to the absolute parent in the traversal so that time complexity would be reduced.

Union by Rank:

In the naive union algorithm we directly point one of the trees to the other but the cost for find operation would be more if we link a big tree consisting of many nodes to a tree having small nodes which increases the height of the tree and costs more for find operation after this union. So we will be using a parameter called rank while doing the union operation.

So now for unioning two trees 1 and 2 we check their ranks if rank of tree 1 is more than rank of tree 2 then tree 2 is made to point to tree 1 else tree 1 is made to point to tree 2 inorder to reduce the height of the resulting tree.

Improved Code:

UnionbyRank_PathCompression.cpp

Section 4.3 - Tree Queries

In this section we discuss some of the common tree queries and their algorithms as well including the Lowest common ancestor,Diameter of a tree,Finding Kth ancestor and rest of the important queries are kept as practice problems at the end of the chapter.

Query: Diameter of a tree

Diameter:

The diameter of a tree is the number of nodes on the longest path between two leaves in the tree.

Algorithm:

- Take some arbitrary node as the root node for processing sub trees.
- Using Dynamic Programming we need to calculate two values for each node x i.e ToLeaf(x) indicating the maximum length of path from node to a leaf and MaxLength(x) indicating the maximum length of path whose highest node is x.
- ToLeaf(x) can be obtained from children of x by adding 1 to child having maximum Toleaf() value and similarly Maxlength(x) can be found using the summation of first two children in decreasing order of Maxlength(x) value with 2.

Code:

Diameter.cpp

Query: Lowest common ancestor

Lowest Common Ancestor:

The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants.

Algorithm:

- Find a path from the root to a given node and store it in a vector or array.
- Find a path from the root to the other node and store it in another vector or array.
- Traverse both paths till the values in arrays are the same. We need to return the common element just before there is mismatch.

Code:

Lowest_Ancestor.cpp

Section 4.4- Problems

Some good problems taken from various coding platforms based on graph algorithms discussed in this chapter are given below.Give them a try!!

1. Collection of Problems
2. XOR-MST
3. K-th ancestor of a node in Binary Tree
4. Entry
5. Find me
6. New Island

CHAPTER 5

SPANNING TREE

Section 5.1- Introduction

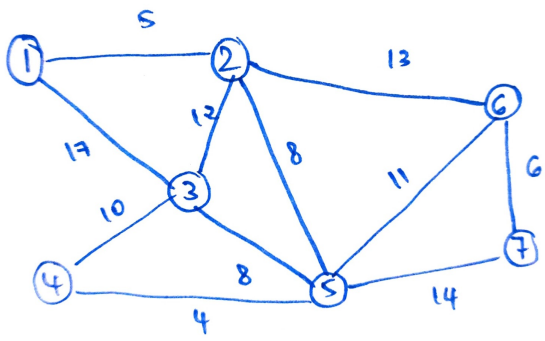
In this chapter we discuss two famous greedy algorithms namely Kruskal algorithm and Prims algorithm that find a minimum spanning tree in the given graph.Working of each of the algorithm is explained using an example and we also look at the code implementation of both the algorithms.At the end of the chapter some of the applications are discussed and problems are provided.

Spanning Tree:
A spanning tree is a subgraph of a given graph G that covers all nodes with a minimum number of edges.

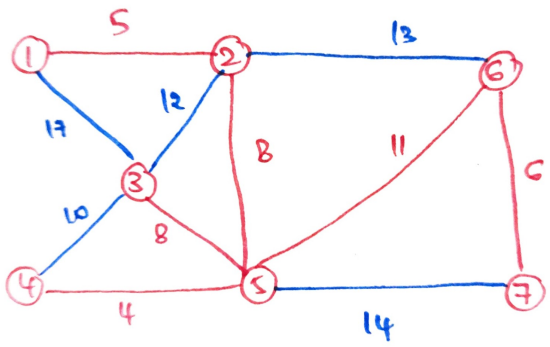
- A spanning tree doesn't have any cycles
- All vertices of a spanning tree are connected.
- The number of edges present in a spanning tree is n-1 where n is number of nodes

Minimum Spanning Tree:
Minimum spanning tree is a spanning tree that has minimum weight among all spanning trees possible.

Example:
Consider a graph G given below,



The corresponding minimum spanning tree (indicated with red color) is given below and it has an edge weight of 42 that covers all the vertices of the given graph.



Section 5.2- Kruskal's Algorithm

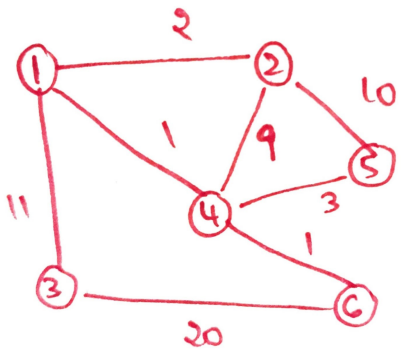
In this section we discuss Kruskal's algorithm and understand it using an example and look at the code part as well and analyze the complexity of the algorithm.

Algorithm:

- We store the graph in the form of an edge list (one of the graph representations discussed in chapter-1) and sort the list in increasing order of their edge weights and initially there are no edges between the nodes.
- Now we add an edge to the subgraph from the edges list if this edge doesn't create a cycle in the new subgraph that's formed.
- Repeat the above step till the total number of edges obtained in the above subgraph is n-1 where n is the number of nodes.This is the required minimum spanning tree.

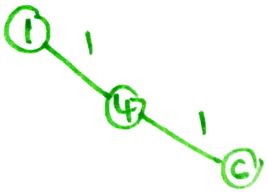
Example:

Consider the below undirected weighted graph

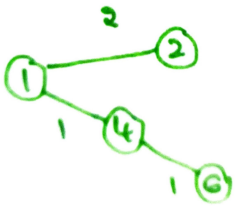


First we sort the edges in the edge list and the least weight edge is between 1 and 4 and 1 and 6, we add one of them, let us say edge between 1 and 4 to the MST.

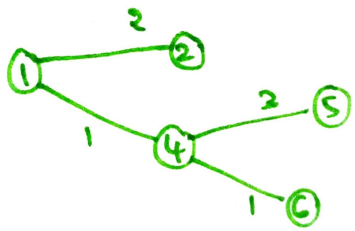
In the next step we add an edge between 4 and 6 to the MST since no cycle is formed upon adding this.



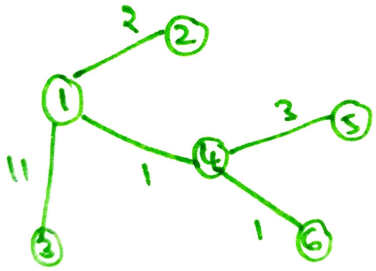
Now the next least weight edge is edge between 1 and 2 and we add it to the MST since it doesn't form cycle



Next we add edge between 4 and 5 as it has next least weight ‘3’



The next least weight edges is 9 which is between 2 and 4 and adding this edge forms a cycle in the graph so we skip adding this edge and similarly with the case of 2 and 5 it will form a cycle.So next we add edge between 1 and 3 and now we are done since all vertices are covered and the resulting graph is the required Minimum Spanning Tree.



Code Implementation:

1. **Pseudo code:**

```
Disjoint_Set DS
sort(edgelist)
wt=0,max=0
for edges in edgelist[]:
  if max==vertices-1
  break
  if find_parent(edge.u)!=find_parent(edge.v)
     Union(edge.u,edge.v)
     wt+=edge.wt;
     max++;

return wt
```

2. **Code:**
   Kruskal.cpp

3. **Complexity:**
   Time Complexity: O(elogn+eloge)
   Sorting of edges takes complexity of O(eloge) and for processing each edge i.e detect formation of cycle since we used Union by rank and path compression this takes O(logv) in the worst case to detect cycle so for processing all edges it would take O(elogv) time.So overall the time complexity of Kruskal algorithm is O(elogn+eloge)

   Space Complexity: O(n+e)
   As we used Disjoint set data structure it needs O(n) space to store the nodes and O(e) while processing the edges.So overall time complexity would be O(n+e).

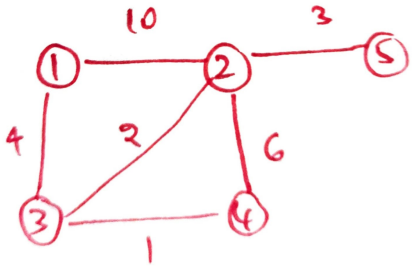
Section 5.3- Prim's Algorithm

In this section we discuss Prim's algorithm and understand it using an example and look at the code part as well and analyze the complexity of the algorithm.

Algorithm:

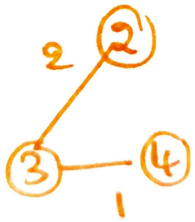
- Initially pick any node from the set of nodes in a graph and place it in a set that contains all the nodes included in the minimum spanning tree.
- Add a minimum cost edge between the set of nodes present in MST and the set of nodes not present in MST and then put the other end of the node to the MST set that is not present earlier.
- Repeat the above step till all the nodes are present in the set.

Example:
Consider the below undirected weighted graph

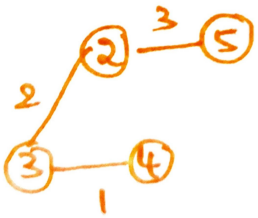


First we pick the edge with least weight i.e edge between 3 and 4 and add nodes 3 and 4 to the set of processed nodes and remove them from unprocessed nodes.

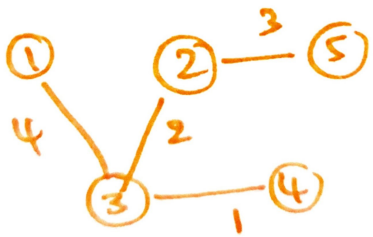
Now we add an edge between 3 and 2 since it is the edge with the least weight that connects a set of processed nodes and unprocessed nodes and add node 2 to the set of processed nodes.



The next least weight edge that connects the set of two nodes is the edge between 2 and 5 and adds node 5 to set of processed nodes.



Now among all the edges connecting the set of nodes the least weight edge is present between node 1 and 3 to our MST and we add node 1 to the processed set and since all the nodes proceed the graph obtained is MST.



Code Implementation:

1. **Pseudo code:**

```
int Prims():

   priority_queue pq;
   wt=0,maxi=0;
   pq.push({0,1});
   initialise mst(n+1,0);
   initialise key(n+1,INFI);
   key[1]=0;

   while(maxi!=n)
       top=pq.top();
       pq.pop();
       wt+=top.first;
       maxi++;
       mst[top.second]=1

       for adj_node in vec[top.second]:
           int node=it.first;
           int wt=it.second;

           If mst[node]==0 and key[node]>wt:
               key[node]=wt;
               pq.push({wt,node});

   return wt;
}
```

2. **Code**:

   Prims.cpp

3. **Complexity:**
   Time complexity: O(elogn+nlogn)
   As we delete all n nodes from the min-heap for getting MST it would take O(nlogn) since each pop operation takes O(logn) and in the worst case we may add E edges and it takes O(Elogv) time since each addition takes O(logv) in min-heap.Hence the overall complexity is O((n+e)loge).
   Space complexity :O(n+e)
   We maintain an array to check if a node is present in MST or not which consumes O(n) and the array to maintain min heap requires O(e) so the overall space complexity is O(n+e).

Section 5.4 - Applications

Some of the applications of Minimum Spanning Tree used in real world algorithms are discussed below:

Network Design:Some common examples are telephone,cable TV,roads,etc...where we want to reduce the total amount of measurable quantity that connects two things in the network like the cable wires that connect TVs in an area,then the amount of wiring could be reduced by modelling the problem as a graph and finding MST accordingly.

Approximation Algorithms:For NP hard problems like Travelling salesman problem,Steiner tree problem the MST could be used for approximation of the values in the algorithms.

Indirect applications:The MST is used in the procedure of many algorithms and hence it has many indirect applications like the problem of max bottleneck path,learning features in face verification in real time and also cluster analysis where the group of clusters is treated as MST and proceeds further to delete nodes.

Section 5.5- Problems

Some good problems taken from various coding platforms based on graph algorithms discussed in this chapter are given below.Give them a try!!

1. GCD and MST
2. XOR-MST
3. Road Reparation
4. Find Imposters
5. Minimum MST Graph

CHAPTER 6

SHORTEST PATHS

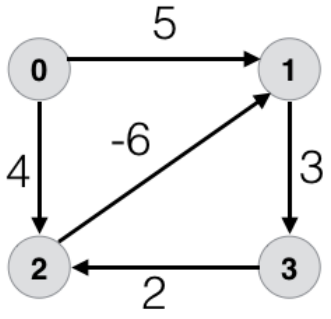
Section 6.1 - Introduction

In this chapter we will be discussing basic algorithms like Dijkstra,Bellman Ford used to find the shortest path between two nodes in a graph and Floyd warshall algorithm to find all pairs shortest paths and look at the code implementation of all three algorithms and also know their drawbacks for specials cases like negative edge weight and negative edge cycles.In the later sections of the book we will be looking at applications and some problems based on the algorithms discussed.

Shortest path problem:
The shortest path problem is the problem of finding a shortest path between two nodes in a graph such that the sum of weights of edges that constitute the path is the least.

Negative Weight Edge:
An edge that has negative weight is called a negative weight edge.

Negative Weight Cycle:
A negative weight cycle is a cycle that has the sum of the weights of the edges constituting as negative.
Example:
Consider the below graph which has a cycle- 1,3,2,1 and the sum of weights in the cycle is -1 so the below graph has a negative weight cycle.



Section 6.2 - Dijkstra Algorithm

Dijkstra Algorithm finds the shortest path from a starting node to the rest of all the nodes.

Algorithm:

- Assign node weights(Node weight denotes the distance of the given node from a source) to all the elements as infinity except the source node.
- Maintain a set of all processed nodes.
- Follow below steps until all the vertices are processed:
    1. Choose the vertex with the minimum value of the node weight.
    2. Add the node to the set of processed nodes.
    3. Update the node weights of all the nodes adjacent to this node if the new weight obtained is less than previous weight of the node.

Code Implementation:

1. **Pseudo code:**

```
void Dijkstra(int source):
init processed_nodes[processed_nodes]=0
init node_weights[num_nodes]={INFINITY}
priority_Queue pq;
 while(num(processed_nodes)!=num_nodes):
   Top_Node=pq.top()
   processed_nodes[Top_Node]=1

   for node in children[Top_Node]:
     if weight(node)>weight(Top_node)+Edge_wt
        weight(node)=weight(Top_node)+Edge_wt

 return
```

2. **Code:**
   Dijkstra.cpp

3. **Analysis:**
   Time complexity: O((n+e)logn)
   Each insertion and deletion operation would take O(logn) and we could see that the loop present in the code could be executed at max n+e times so complexity would be O((n+e).logn)).
   Space complexity: O(n+e)
   The code uses a vector for storing the node weights and processed nodes so that it would take an order of O(n+e).

Section 6.3 - Bellman Ford Algorithm

Bellman Ford Algorithm finds the shortest path from a starting node to the rest of all the nodes and it is prefered over Dijkstra in the case of negative weight edges present in the graph as Dijkstra algorithm may or may not work for a graph having negative edge weights.

Algorithm:

1. Assign node weights to all elements as infinity except source node.
2. Repeat the below steps for n-1 times where n is the number of nodes:
    - Traverse the Edge list each time and update the node weights of the nodes of the current edge if the new weight obtained is less than the previous weight of the node.
3. Again update all the node weights and if there is change in node weight of any node it indicates that there is definitely at least one negative edge cycle in the graph.

Code Implementation:

1. **Pseudo code:**

```
void update(edge):
   int start=edge.start,end=edge.end,int wt=edge.weight
   if end==source continue
   if node_wt(v)>node_wt(u)+wt
       node_wt(v)=node_wt(u)+wt
return
void Bellmand_Ford():
 init node_wts = INFINITY
 init node_wts[source_node]=0
 count=num_nodes
while count!=num_nodes:
  count++
  for edge in edge_list[]:
      update(edge)
return
```

2. **Code:**
   Bellman_Ford.cpp
3. **Analysis:**
   Time complexity:O(ne)
   In the worst case we may relax each edge ‘n’ number of times where n is the total number of nodes so resulting time complexity is O(ne).
   Space complexity:O(n+e)
   Since we are using edge list representation to store the edges and node information is stored in an array, the resulting complexity is O(n+e).

Section 6.4 - Floyd Warshall Algorithm

Floyd Warshall Algorithm finds the shortest path from each node to all other nodes.

Algorithm:

1. initialise the result matrix of size n*n where n is the number of nodes and having the values same as that of adjacency matrix representation of the graph.
2. Include each vertex in the shortest path from one vertex to the other if the path with that vertex is shorter than the previous one and update the path accordingly.
3. After checking all vertices if they are present as an intermediate vertex between any two nodes in the shortest path,we get the final result matrix where each entry result[i][j] indicates the distance of shortest path between nodes i and j.

Code Implementation:

1. **Pseudo code:**

```
void Floyd_Warshall(result){
   for(int i=1;i<=n;i++){
       for(int j=1;j<=n;j++){
           for(int k=1;k<=n;k++){
               result[j][k]=min(result[j][k],result[j][i]+[i][k]);
           }
       }
   }
     return;
}
```

2. **Code:**
   Floyd_Warshall.cpp

3. **Analysis:**
   Time complexity: O(n^3)
   In the code we can see 3 nested for loops and each runs in O(n) time so the overall complexity of the algorithm is O(n^3).
   Space complexity: O(n^2)
   Since we maintain a 2D array for maintaining distances of size n*n so the space complexity of the algorithm is O(n^2).

Section 6.5 - Applications

The following are the applications of the shortest path algorithms:

- The algorithms are used In G-Maps to find the shortest path between two locations.
- In social networking websites like facebook,the algorithms are used to suggest friends.
- They are also used in inversion of real matrices
- The algorithms are also widely used to designate file server,Telephone network and social networking applications.

Section 6.6 - Problems

Some good problems taken from various coding platforms based on graph algorithms discussed in this chapter are given below.Give them a try!!

1. The village
2. Creating Offices
3. Alice in forest
4. Tree Distances I
5. Palace

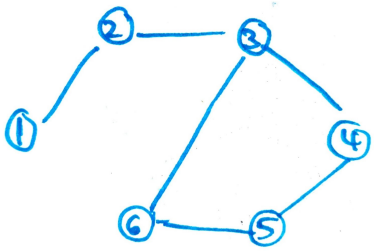
CHAPTER 7

CONNECTIVITY

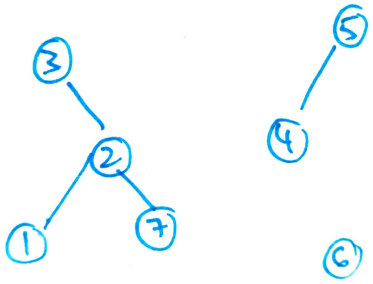
Section 7.1 - Introduction

In this chapter we will be looking at two fundamental algorithms namely Kosaraju algorithm and Tarjan's algorithm to find the strongly connected components in the graph and we look at the code implementation of the algorithm as well and discuss the applications of the algorithms and try to solve the problems present at the end of the chapter.

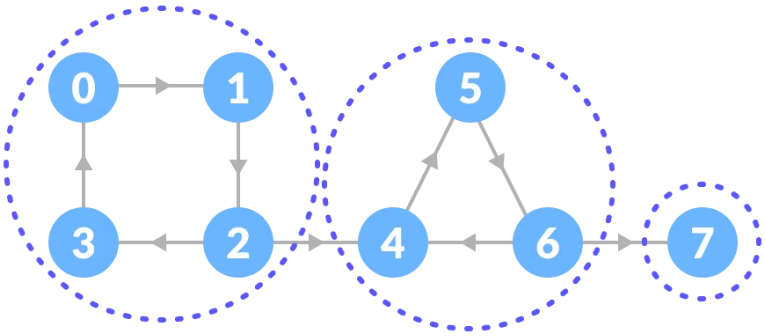
Connected Graph:
Connected Graph is a graph that has a path between a pair of vertices i.e there are no unreachable vertices.
Example:



Disconnected Graph:
The graph which is not connected is called a disconnected graph.
Example:



Strongly Connected components:
A strongly connected component of a directed graph G is a maximal set of vertices such that for every pair of vertices u and v, there is a directed path from u to v and a directed path from v to u.
Example:



Section 7.2 - Kosaraju Algorithm

Kosaraju algorithm is one of the algorithms used to find the strongly connected components in a graph.In this section we will be looking at the algorithm and the code implementation as well.

Algorithm:

1. Perform a DFS traversal on the graph and push the elements into the stack before they return from the function.
2. Transpose the graph by reversing the direction of edges and this is the reverse graph needed in the next step.
3. Pop the elements out of the stack by performing DFS as well and each DFS would give a strongly connected component before popping out the next element which is not part of the current component.

Code Implementation:

1. **Pseudo Code:**

```
void Kosaraju():
   //step-1:perform DFS and push to stack accordingly
   init visited[n]=0
   for(int i=1;i<=n;i++){
       if visited(i)==false
       DFS(i,visited);//pushes elements to stack before returning
   }
   init visited[n]=0
   //step-2:reverse the graph
   reverse_graph(); //just change edges direction
   //step-3:perform DFS and separate out the SCC
   while(!empty(stack))
     if(!visited[top(stack)])
       vector<int>use;
       DFS2(top(stack),visited,use);//pushes elements before calling
       scc.push_back(use);
     s.pop();
   return
```

2. **Code:**

   Kosaraju.cpp

3. **Analysis:**

   Time complexity: O(n+e)

   As we do DFS on the whole graph two times the complexity would be the same as the time complexity of DFS i.e O(n+e).

   Space complexity: O(n)

   We are maintaining an array to keep track of the visited and processed that are taking space of O(n).

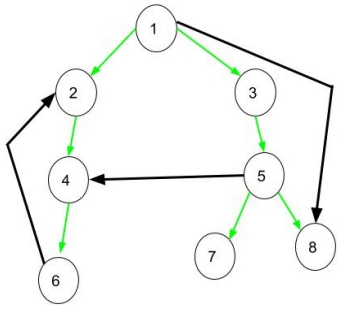
Section 7.3 - Tarjan's Algorithm

Tarjan's algorithm is one of the algorithms used to find the strongly connected components in a graph.In this section we will be looking at the algorithm and the code implementation as well.

Prerequisites:

Tree Edge:
It is an edge which is present in the tree obtained after applying DFS on the graph.
Forward Edge:
It is an edge (u, v) such that v is descendant but not part of the DFS tree.
Back edge:
It is an edge (u, v) such that v is the ancestor of node u but not part of the DFS tree.
Cross Edge:
It is an edge which connects two nodes such that they do not have any ancestor and a descendant relationship between them.

Let us consider below example to understand the terms



- All the Green edges are tree edges.
- Edge from 1 to 8 is a forward edge.
- Edge from 6 to 2 is a back edge.
- Edge from node 5 to 4 is cross edge.

Low:
The Node with lowest discovery time that is accessible.
Discovery time:
This is the time when a node is visited 1st time while DFS traversal.

Algorithm:

1. Maintain a timer that would increase the value by 1 unit each time it discovers a new node.
2. Apply DFS on each node if it is not visited earlier and assign the corresponding discovery time and low time.
3. Whenever you find a node to be visited in DFS traversal then the connecting edge is either back edge or cross edge.To differentiate between both of them maintain a stack of nodes in current DFS traversal and if the node is present in the stack then it's a back edge else it is a cross edge.
4. For back edges as mentioned earlier the value of low time while backtracking is low[u]=min(low[u],disc[v]) and we don't process the cross edge since it is not required.
5. While backtracking in DFS the low time would be updated as low[u]=min(low[u],low[v]) for edge u directed to v and backtracking from v to u.
6. Whenever the value if low and discover are same in the stage of backtracking for any node then it is considered as the head node and elements are popped out of the stack till this head node is reached and this set of elements popped out along with the head node is strongly connected.
7. After all backtracking the stack would be empty and the resulting SCC are either stored or printed.

Code Implementation:

1. **Pseudo Code:**

```
void DFS(int i,stack<int> &s,vector<vector<int>> &info){
   info[i][0]=timer;
   info[i][1]=timer;
   info[i][2]=1;
   timer++;
   s.push(i);
   for(auto it:vec[i]){
       if(info[it][0]==-1){//element not yet discovered
          DFS(it,s,info);
          info[i][1]=min(info[it][1],info[i][1]);
       }

       else if(info[it][2]==1){ //for back edge
           info[i][1]=min(info[it][0],info[it][1]);
       }
       //we ignore cross edge processing
   }
   if(info[i][0]==info[i][1]){//head node
       vector<int>use;
       use.push_back(i);
       while(s.top()!=i){
           int top=s.top();
           use.push_back(top);
           info[top][2]=0;//popped out from stack
           s.pop();
       }
       info[i][2]=0;
       s.pop();
       scc.push_back(use);
   }
   return;
}
```

```
void Tarjan(){
   vector<vector<int>>info(n+1,{-1,-1,0});//stores disc time,low
time,Is_in_stack
   stack<int>s;//to store elements in dfs traversal

   for(int i=1;i<=n;i++){
       if(info[i][0]==-1)
       DFS(i,s,info);
   }

   for(auto it:scc){
       for(auto it2:it){
           cout<<it2<<" ";
       }
       cout<<endl;
   }
   return;
}
```

2. **Code**:
   Tarjan.cpp

3. **Analysis:**
   Time complexity: O(n+e)
   As we use modified DFS in the algorithm to update low and disc time for all the nodes the complexity would be the same as the time complexity of DFS i.e O(n+e).
   Space complexity:
   We are maintaining an array to keep track of the visited and processed that are taking space of O(n).

Section 7.4 - Applications

The strongly connected component algorithms can be used as a first step in many graph algorithms that work only on strongly connected graph.In social networks, a group of people are generally strongly connected like students of a class or any other common place and the same is used by apps like facebook that suggest some common things between two people. Many people in these groups generally like some common pages or play common games. The Strongly connected component algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked a commonly liked page or played a game.

Section 7.5 - Problems

Some good problems taken from various coding platforms based on graph algorithms discussed in this chapter are given below.Give them a try!!

1. Frog Traveller
2. Creating Offices
3. Path Queries
4. Subtree Queries
5. Overload -our

CHAPTER 8

FLOWS AND CUTS

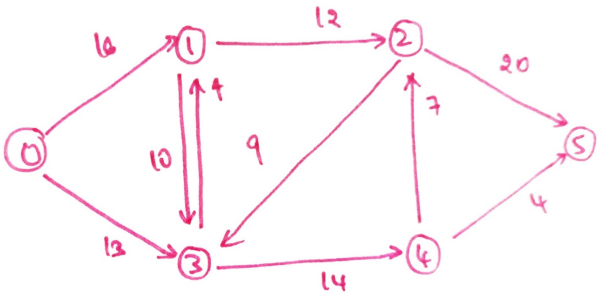
Section 8.1 - Introduction

In this chapter we will be looking at an algorithm namely Ford-Fulkerson algorithm to find the maximum flow that can be sent from a source to a destination in the graph and also look at the relation between maximum flow and minimum cut in a graph and then we look at the code implementation of the algorithm as well and discuss the applications of the algorithms and try to solve the problems present at the end of the chapter.

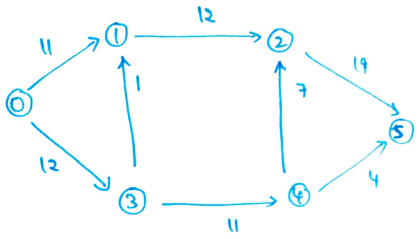
Maximum Flow Problem:
What is the maximum flow that we can send from one node to the other in a graph?Suppose in a given graph that represents a flow network where every edge has some capacity. Also given two nodes source s and sink t in the graph we need to find the maximum possible flow from s to t with following constraints:

1. Flow on an edge doesn't cross the given capacity of the edge.
2. Incoming flow is equal to outgoing flow for every vertex except s and t.

Let us consider the below graph which is a flow network whose source is 0 and sink is 5.



The maximum flow in the graph above is 23 (11+12) as we could see that the flow can be sent in the below way:



Source:
A vertex whose degree is 0 and it is the start of all outgoing edges is called a source.

Sink:
A vertex whose degree is 0 and it is the end of all outgoing edges is called a source.

Section 8.2 - Ford Fulkerson Algorithm

The Ford Fulkerson algorithm is one of the algorithms used to find the maximum flow in a network that can be sent from a source node to destination node in a graph.In this section we will be looking at the algorithm and the code implementation as well.

Algorithm:

1. Create a copy of the graph known as residual graph to update the values of residual edge weights each time and initialise max flow value to 0.
2. Find a path from source to sink using BFS traversal.
3. Find the minimum edge weight w for that path and update the values for forward edge by decrementing with w and backward edges by incrementing with w and increment the max flow value accordingly.
4. Follow the above step until there exists no path from source to sink in the updated graph.The resultant value present in max flow is the required maximum flow.

Code Implementation:

1. **Pseudo Code:**

```
int Ford_Fulkerson(int s,int t)
   int max_flow=0;

   while(BFS(s,t,parent))
       int path_flow=INFI
       int v=t;
       while(v!=s){
           int u=parent[v]
           path_flow=min(path_flow,rvec[u][v]
           v=u
       v=t
       //updating the values each time
       while(v!=s)
           int u=parent[v];
           rvec[u][v]-=path_flow
           rvec[v][u]+=path_flow
           v=u;

       max_flow+=path_flow
   }

   return max_flow
```

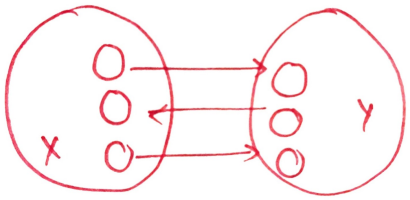
2. **Code:**
   Ford_Fulkerson.cpp
3. **Analysis:**
   Time complexity: O(e.n^2)
   We find the path from source to sink each time using BFS as it finds the minimum number of edges in tha path each time so BFS takes time complexity of O(n^2) as we are using adjacency matrix representation and in the worst case it could pick all the edges so time complexity goes to O(e.n^2).

Space complexity: O(n)
We are maintaining an array to keep track of the visited and processed for the BFS called each time that is taking space of O(n).

Section 8.3 - Relation between Max Flows and Min Cut

Minimum Cut Problem:
We need to find the minimum-weight set of edges that separates two nodes of the graph.We shall remove a set of edges from the graph such that there will be no path from the source to the sink after the removal and the total weight of the removed edges is minimum.

It turns out that once the Ford–Fulkerson algorithm has found a maximum flow,it has also determined a minimum cut. The flow produced by the algorithm maximum and hence is the cut minimum.Let A be the set of nodes that can be reached from the source using positive-weight edges. The reason is that a graph cannot contain a flow whose size is more than the weight of any cut of the graph. Hence, always when a flow and a cut are equally large, they are a maximum flow and a minimum cut.



Let us consider any cut of the graph such that the source belongs to A , the sink belongs to B and there are some edges between the sets:
The size of the cut is the sum of the edges that go from A to B . This is an upper bound for the flow in the graph, because the flow has to proceed from A to B. Thus, the size of a maximum flow is smaller than or equal to the size of any cut in the graph.

Section 8.4 - Applications

Ford Fulkerson is one of the famous algorithms for finding the maximum flow in a network that can be sent from one source node to the other sink node and also we have looked at the residual graph and residual capacities and also have understood that the minimum cut in a graph is same as the maximum flow in a graph and looked at the theoretical proof as well.This kind of the problems is usually common in the real world scenarios like the problem of maximizing the transportation with the given limited number of traffic lights and maximuming the flow in case of computer networks and many such real applications are present for this algorithm.

Section 8.5 - Problems

Some good problems taken from various coding platforms based on graph algorithms discussed in this chapter are given below.Give them a try!!

1. Unknown enemy
2. Weird Watch
3. Road Reparation
4. Find Imposters
5. Desert
6. Array Stabilization

THE END

Hope you enjoyed learning!!