IEEEXtreme Türkiye Camp 24' Program

Day 3

1 Introduction to Graph

A graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to the mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge. Typically, a graph is depicted in diagrammatic form as a set of dots for the vertices, joined by lines for the edges. [1]

Why graphs? Graphs are usually used to represent different elements that are somehow related to each other.

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

G = (V,E)

V = set of nodes

E = set of edges(e) represented as e = a,b

Graph are used to show a relation between objects. So, some graphs may have directional edges (e.g. people and their love relationships that are not mutual: Alice may love Alex, while Alex is not in love with her and so on), and some graphs may have weighted edges (e.g. people and their relationship in the instance of a debt)

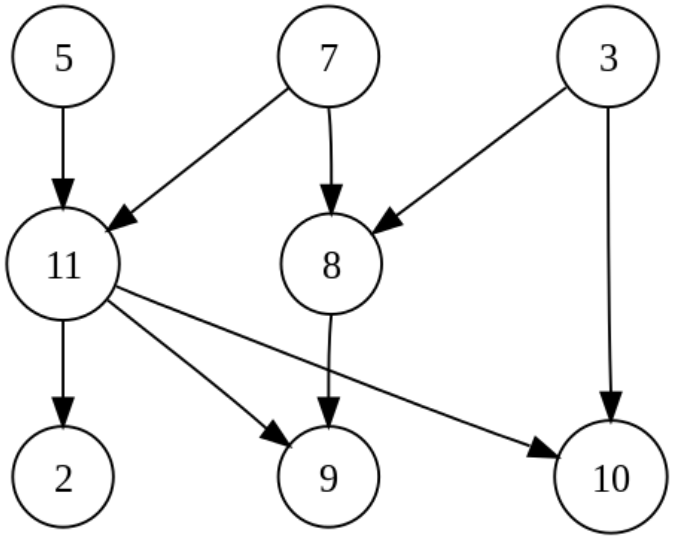

Figure 1: a simple unweigted graph

2 Definitions

2.1 Definitions of Common Terms

- `Node` - An individual data element of a graph is called Node. Node is also known as vertex.
- `Edge` - An edge is a connecting link between two nodes. It is represented as e = a,b Edge is also called Arc.
- `Adjacent` - Two vertices are adjacent if they are connected by an edge.
- `Degree` - a degree of a node is the number of edges incident to the node.
- `Undirected Graphs` - Undirected graphs have edges that do not have a direction. The edges indicate a two-way relationship, in that each edge can be traversed in both directions.
- `Directed Graphs` - Directed graphs have edges with direction. The edges indicate a one-way relationship, in that each edge can only be traversed in a single direction.
- `Weighted Edges` - If each edge of graphs has an association with a real number, this is called its weight.
- `Self-Loop` - It is an edge having the same node for both destination and source point.
- `Multi-Edge` - Some Adjacent nodes may have more than one edge between each other.

2.2 Walks, Trails, Paths, Cycles and Circuits

- `Walk` - A sequence of nodes and edges in a graph.
- `Trail` - A walk without visiting the same edge.
- `Circuit` - A trail that has the same node at the start and end.
- `Path` - A walk without visiting same node.
- `Cycle` - A circuit without visiting same node.

2.3 Special Graphs

- `Complete Graph` - A graph having at least one edge between every two nodes.
- `Connected Graph` - A graph with paths between every pair of nodes.
- `Tree` - an undirected connected graph that has any two nodes that are connected by exactly one path. There are some other definitions that you can notice it is tree:
  - an undirected graph is connected and has no cycles. an undirected graph is acyclic, and a simple cycle is formed if any edge is added to the graph.
  - an undirected graph is connected, it will become disconnected if any edge is removed.
  - an undirected graph is connected, and has (number of nodes - 1) edges.

3 Representing graphs

3.1 Edge lists

A simple way to define edge list is that it has a list of pairs. We just have a list of objects consisting of the vertex numbers of 2 nodes and other attributes like weight or the direction of edges. [17]

- + For some specific algorithms you need to iterate over all the edges, (i.e. kruskal's algorithm)
- + All edges are stored exactly once.
- − It is hard to determine whether two nodes are connected or not.
- − It is hard to get information about the edges of a specific vertex.

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    int edge_number;
    vector<pair <int,int> > edges;
    cin » edge_number;
    for( int i=0 ; i<edge_number ; i++ ){
        int a,b;
        cin » a » b;
        edges.push_back(make_pair(a,b)); // a struct can be used if edges are weighted or
    }
}
```

3.2 Adjacency Matrices

Stores edges, in a 2-D matrix. matrix[a][b] keeps an information about road from a to b. [17]

- + We can easily check if there is a road between two vertices.
- − Looping through all edges of a specific node is expensive because you have to check all of the empty cells too. Also these empty cells takes huge memory in a graph which has many vertices.(For example representing a tree)

```
    #include <iostream>
    #include <vector>
    using namespace std;
    int main(){
        int node_number;
        vector<vector<int> > Matrix;
```

```
            cin » node_number;
            for( int i=0 ; i<node_number ; i++ )
                for( int j=0 ; j<node_number ; j++ ){
                    Matrix.push_back(vector <int> ());
                    int weight;
                    cin »weight ;
                    Matrix[i].push_back(weight);
                }
        }
```

3.3 Adjacency List

Each node has a list consisting of nodes each is adjacent to. So, there will be no empty cells. Memory will be equal to number of edges. The most used one is in algorithms. [17]

- + You do not have to use space for empty cells,
- + Easily iterate over all the neighbors of a specific node.
- − If you want to check if two nodes are connected, in this form you still need to iterate over all the neighbors of one of them. But, there are some structures that you can do this operation in O(log N). For example if you won't add any edge, you can sort every vector with nodes' names, so you can find it by binary search.

```
        #include <iostream>
        #include <vector>
        using namespace std;

        int main(){
            int node_number,path_number;

            vector<vector<int> > paths;
            // use object instead of int,
            //if you need to store other features

            cin » node_number » path_number;
            for( int i=0 ; i<node_number ; i++ )
                Matrix.push_back(vector <int> ());
            for( int j=0 ; j< path_number ; j++ ){
                int  beginning_node,end_node;
                cin » beginning_node » end_node;

                Matrix[ beginning_node ].push_back( end_node ); // push st
                // Matrix[ end_node ].push_back(  beginning_node );
                // ^^^ If edges are Undirected, you should push in reverse direction too
            }
        }
```

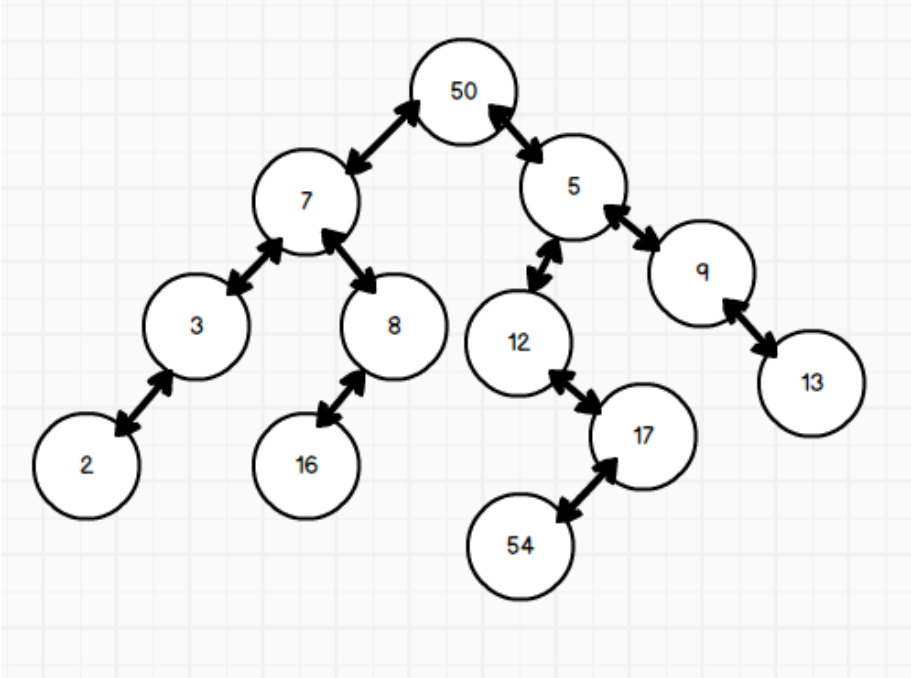

Figure 2: a binary tree

4 Tree Traversals

The tree traversal is the process of visiting every node exactly once in a tree structure for some purposes(like getting information or updating information). In a binary tree there are some described order to travel, these are specific for binary trees but they may be generalized to other trees and even graphs as well. [2]

4.1 Preorder

Preorder means that a root will be evaluated before its children. In other words the order of evaluation is: Root-Left-Right

- Preorder
- 1. Look Data
- 2. Traverse the left node
- 3. Traverse the right node
- Example: 50 7 3 2 8 16 5 12 17 54 9 13

4.2 Inorder

Inorder means that the left child (and all of the left childs children) will be evaluated before the root and before the right child and its children. Left-Root-Right (by the way, in binary search tree inorder retrieves data in sorted order)

- Inorder
- 1. Traverse the left node
- 2. Look Data
- 3. Traverse the right node
- Example: 2 3 7 16 8 50 12 54 17 5 9 13

4.3 Postorder

Postorder is the opposite of preorder, all children are evaluated before their root: Left-Right-Root

- Postorder
- 1. Traverse the left node
- 2. Traverse the right node
- 3. Look Data
- Example: 2 3 16 8 7 54 17 12 13 9 5 50

4.4 Implementation

```
class Node:
    def __init__(self,key):
        self.left = None
        self.right = None
        self.val = key

def printInorder(root):
    if root:
        printInorder(root.left)
        print(root.val),
        printInorder(root.right)

def printPostorder(root):

    if root:
        printPostorder(root.left)

        printPostorder(root.right)

        print(root.val),

def printPreorder(root):

    if root:
        print(root.val),

        printPreorder(root.left)

        printPreorder(root.right)
```

5 Binary Search Tree

A Binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

For a binary tree to be a binary search tree, the values of all the nodes in the left sub-tree of the root node should be smaller than the root node's value. Also the values of all the nodes in the right sub-tree of the root node should be larger than the root node's value. [5, 6, 14]

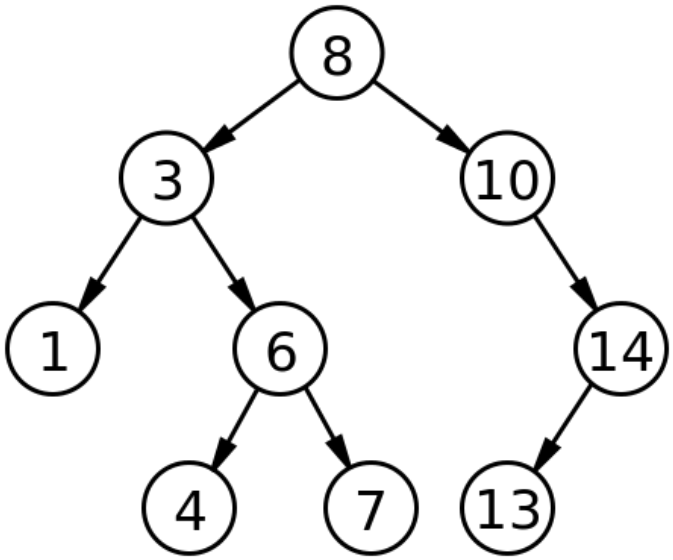

Figure 3: a simple binary search tree

5.1 Insertion Algorithm

- Step 1: Compare values of the root node and the element to be inserted.
- Step 2: If the value of the root node is larger, and if a left child exists, then repeat step 1 with root = current root's left child. Else, insert element as left child of current root.
- Step 3: If the value of the root node is lesser, and if a right child exists, then repeat step 1 with root = current root's right child. Else, insert element as right child of current root.

5.2 Deletion Algorithm

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor.
- Note that: inorder successor can be obtained by finding the minimum value in right child of the node.[15]

5.3 Sample Code

```
// C program to demonstrate delete operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

```

```
37      /* Otherwise, recur down the tree */
38      if (key < node->key)
39          node->left  = insert(node->left, key);
40      else
41          node->right = insert(node->right, key);
42
43      /* return the (unchanged) node pointer */
44      return node;
45  }
46
47  /* Given a non-empty binary search tree, return the node with minimum
48     key value found in that tree. Note that the entire tree does not
49     need to be searched. */
50  struct node * minValueNode(struct node* node)
51  {
52      struct node* current = node;
53
54      /* loop down to find the leftmost leaf */
55      while (current->left != NULL)
56          current = current->left;
57
58      return current;
59  }
60
61  /* Given a binary search tree and a key, this function deletes the key
62     and returns the new root */
63  struct node* deleteNode(struct node* root, int key)
64  {
65      // base case
66      if (root == NULL) return root;
67
68      // If the key to be deleted is smaller than the root's key,
69      // then it lies in left subtree
70      if (key < root->key)
71          root->left = deleteNode(root->left, key);
72
73      // If the key to be deleted is greater than the root's key,
74      // then it lies in right subtree
75      else if (key > root->key)
76          root->right = deleteNode(root->right, key);
77
78      // if key is same as root's key, then This is the node
79      // to be deleted
80      else
81      {
82          // node with only one child or no child
83          if (root->left == NULL)
84          {
85              struct node *temp = root->right;
86              free(root);
87              return temp;
88          }
89          else if (root->right == NULL)
90          {
91              struct node *temp = root->left;
```

```
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

5.4 Time Complexity

The worst case time complexity of search, insert, and deletion operations is O(h) where h is the height of Binary Search Tree. In the worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become N and the time complexity of search and insert operation may become $O(N)$. So the time complexity of establishing N node unbalanced tree may become $O(N^2)$ (for example the nodes are being inserted in a sorted way). But, with random input the expected time complexity is $O(NlogN)$.

However, you can implement other data structures to establish Self-balancing binary search tree (which will be taught later), popular data structures that implementing this type of tree include:

- 2-3 tree
- AA tree
- AVL tree
- B-tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap
- Weight-balanced tree

6 Depth First Search

Depth First Search (DFS) is an algorithm for traversing or searching tree. (For example, you can check if graph is connected or not via DFS) [16]

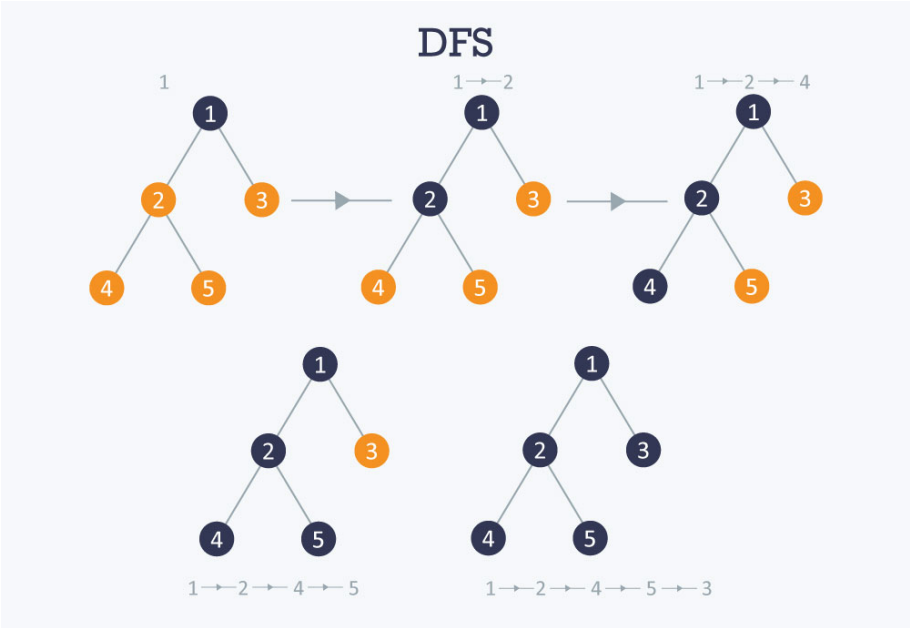

Figure 7: example of dfs traversal

6.1 Method

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to t raverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected. [19]

```
vector<vector<int>> adj; // graph represented as an adjacency list
int n; // number of vertices
vector<bool> visited;
void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
}
```

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows: Pick a starting node and push all its adjacent nodes into a stack. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop. [19]

```
        DFS-iterative (G, s):   //Where G is graph and s is source vertex
      let S be stack
      S.push( s )   //Inserting s in stack
      mark s as visited.
      while ( S is not empty):
          //Pop a vertex from stack to visit next
          v  =  S.top( )
         S.pop( )
         //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                     S.push( w )
                    mark w as visited
```

Example Question: Given an undirected graph, find out whether the graph is strongly connected or not? An undirected graph is strongly connected if there is a path between any two pair of vertices.

```
#include <bits/stdc++.h>
using namespace std;

cont int MaxN=100005; // Max number of nodes

vector <int> adj[MaxN];
bool mark[MaxN];

void dfs(int k)
{
    mark[k]=1;  // visited

    for(auto j : adj[k]) // iterate over adjacent nodes
        if(mark[j]==false) // check if it is visited or not
            dfs(j); // do these operation for that node
}
int main()
{
    cin » n » m;  // number of nodes , number of edges
    for (int i=0 ; i < m; i++)
    {
        cin » a » b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }
    dfs(1);

    bool connected=1;
```

```
29      for(int i=1 ; i <= n ;i++)
30          if(mark[i]==0)
31          {
32              connected=0;
33              break;
34          }
35      if(connected)
36          cout « "Graph is connected" « endl;
37      else
38          cout « "Graph is not connected" « endl;
39      return 0;
40  }
```

6.2 Complexity

The time complexity of DFS is O (V+E) when implemented using an adjacency list (with Adjacency Matrices it is $O(V^2)$), where V is the number of nodes and E is the number of edges. [3]

7 Breadth First Search

Breadth First Search (BFS) is an algorithm for traversing or searching tree. (For example, you can find the shortest path fron one node to another in an unweighted graph.

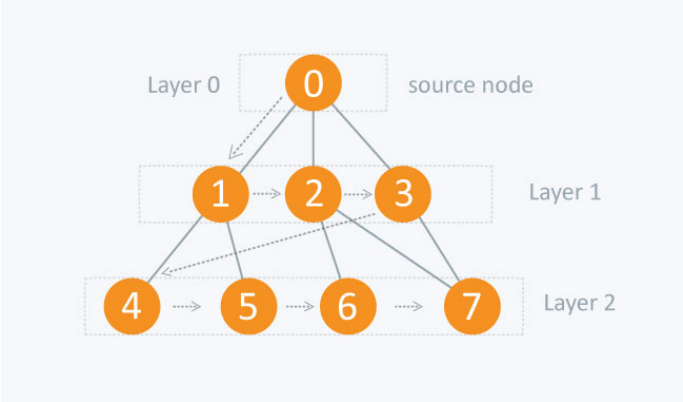

Figure 8: An example breadth first search traversal

7.1 Method

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes. [4]

- As the name BFS suggests, you are required to traverse the graph breadthwise as follows:
- First move horizontally and visit all the nodes of the current layer
- Add to the queue neighbour nodes of current layer.
- Move to the next layer, which are in the queue

Example question: Given a unweighted graph, a source and a destination, we need to find shortest path from source to destination in the graph in most optimal way?

```
#include <bits/stdc++.h>
using namespace std;

cont int MaxN=100005; // Max number of nodes

vector <int> adj[MaxN];
bool mark[MaxN];

void bfs(int starting_point,int ending_point)
```

```
{
    memset(mark,0,sizeof(mark)); //clear the cache
    queue <pair <int,int> > q; // the value of node
    // , and length between this node and the starting node
    q.push_back(make_pair(starting_point,0));
    mark[starting_point]=1;
    while(q.empty()==false)
    {
        pair <int,int> tmp = q.front(); // get the next node
        q.pop(); // delete from q
        if(ending_point==tmp.first)
        {
            printf("The length of path between %d - %d : %d\n",
            starting_point,ending_point,tmp.second);
            return ;
        }
        for (auto j : adj[tmp.first])
        {
            if(mark[j]) continue ; // if it reached before
            mark[j]=1;
            q.push_back(make_pair(j,tmp.second+1)); // add next node to queue
        }
    }
}
int main()
{
    cin » n » m;  // number of nodes , number of edges
    for (int i=0 ; i < m; i++)
    {
        cin » a » b;
        adj[a].push_back(b);
    }
    cin » start_point » end_point;
    bfs(start_point);

    return 0;
}
```

7.2 Complexity

The time complexity of BFS is O(V + E), where V is the number of nodes and E is the number of edges.

8 Shortest Path Problem

8.1 Definition

Let $G(V, E)$ be a graph, v_i and v_j be two nodes of G. We say a path between v_i and v_j is the shortest path if sum of the edge weights (cost) in the path is minimum. In other words, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. [7]

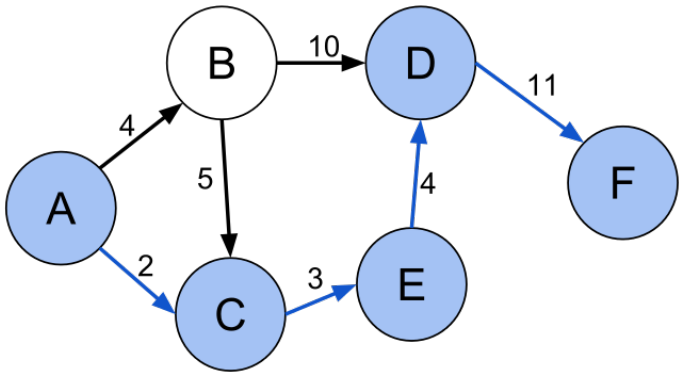

Figure 6: Example shortest path in graph. Source is A and target is F. Image taken from [7].

We will cover several shortest path algorithms in this bundle. One of them is Dijkstra's Shortest Path Algorithm but it has some drawbacks: Edge weights should be non-negative for the optimally of the algorithm. We will discover other algorithms in which these condition isn't necessary, like Floyd-Warshall and Bellman-Ford algorithms.

8.2 Dijkstra's Shortest Path Algorithm

Dijkstra's Shortest Path algorithm is straight forward. In brief we have a set S that contains explored nodes and d which contains the shortest path cost from source to another node. In other words, $d(u)$ represents the shortest path cost from source to node u. The procedure follows as that. First, add source node to set S which represents the explored nodes and assigns the minimum cost of the source to zero. Then each iteration we add node to S that has lowest cost ($d(u)$) from unexplored nodes. Let's say $S' = V - S$ which means unexplored nodes. For all nodes in S' we calculate $d(x)$ for each node x is S' then we pick minimum cost node and add it to S. So how we calculate $d(x)$?. For any x node from S', $d(x)$ calculated as that, let's say e cost of any edge from S to x then $d(x) = min(d(u) + e)$. It is a greedy algorithm.

Here is the explanation of the algorithm step by step.

1. Initialize an empty set, distance array, insert source to set.
2. Initialize a min-heap, put source to heap with key is zero.
3. While heap is not empty, take the top element from heap and add its neighbours to min-heap.
4. Once we pick an element from the heap, it is guaranteed that the same node will never be added to heap with lower key value.

In implementatio,n we can use priority queue data structure in order to increase efficiency. If we put unexplored nodes to min - priority queue where the distance is key, we can take the lowest cost unexplored node in $O(log(n))$ time which is efficient.

```
typedef pair<int,int> edge;
typedef vector<edge> adjList;
typedef vector<adjList> graph;

void dijkstra(graph &g, int s){
    vector<int> dist(g.size(),INT_MAX/2);
    vector<bool> visited(g.size(),false);
    dist[s] = 0;
    priority_queue<edge, vector<edge>, greater<edge>> q;
    q.push({0, s});
    while(!q.empty())
    {
        int v = q.top().second;
        int d = q.top().first;
        q.pop();
        if(visited[v]) continue;
        visited[v] = true;
        for(auto it: g[v])
        {
            int u = it.first;
            int w = it.second;
            if(dist[v] + w < dist[u])
            {
                dist[u] = dist[v] + w;
                q.push({dist[u], u});
            }
        }
    }
}
```

8.3 Floyd-Warshall Algorithm

The Floyd Warshall Algorithm is used for solving the all pairs shortest path problem. The problem is to find the shortest shortest distances between every pair of vertices in a given weighted directed graph [8]. Instead of running Dijkstra's algorithm for every node as a source, Floyd-Warshall algorithm provides a simpler solution that uses the power of dynamic programming to achieve this task.

Lets state that in this algorithm, we have adjacency matrix representation of the graph. This algorithm works optimal even if there are negative edges but not negative cycles unlike the Dijkstra's shortest path. The algorithm looks for if any node can be an intermediate node for a path that decrease the cost. If the new cost is smaller, algorithm updates the cost in the adjacency matrix. For every k (as an intermediate node) in graph, we check all i,j pairs and we calculate $cost(i, k)+cost(k, j)$. Then we update $cost(i, j)$ with the new value if it is smaller than the current value.

```
void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
      distances between every pair of vertices */
    int dist[V][V], i, j, k;
    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];
    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

Time complexity of the algorithm can be seen here, there are three nested for loops over all nodes hence it is $O(V^3)$ and space complexity is $O(V^2)$ because we keep adjacency matrix in memory(after altering it with new values, it becomes a memoization table).

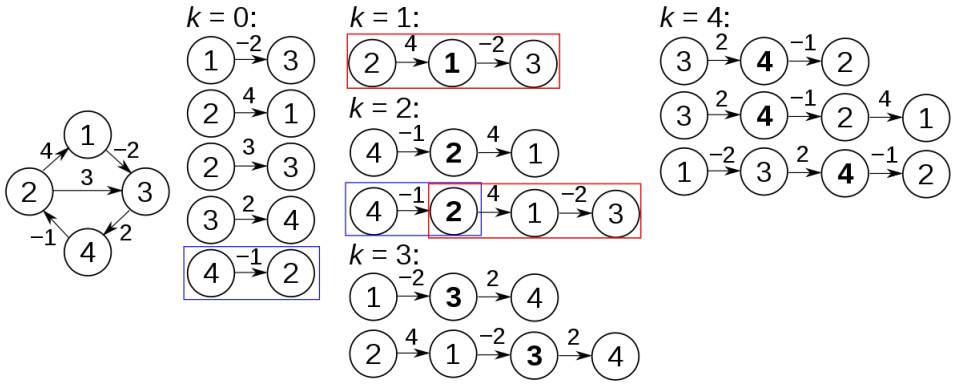

Figure 7: Example of Floyd-Warshall

8.4 Bellman Ford Algorithm

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. Bellman-Ford algorithm also detects negative weighted cycles in the graph [9].

In this algorithm, we maintain distance and the previous arrays to save costs and path. This algorithm is for single source shortest path problem and we initialize distance[source] to zero and all others to infinity. Then we check for all vertices in graph with all edges if using this edge makes the distance smaller than the current distance to this node. If it is smaller, we update distance and previous arrays. After constructing the previous and distance arrays, we loop over the edges and if we find a change on distance array, it means that graph contains a negative cycle.

```
typedef pair<int,pair<int,int>> edge;
typedef vector<edge> weigthed_graph;
void BellmanFord(weigthed_graph g, int V, int src)
{
    int E = g.size();
    int dist[V];
    //this implemantation is just for negative cycle check
    // init distance array
    for (int i = 0; i < V; i++)
        dist[i]   = INT_MAX;

    dist[src] = 0;
    for (int i = 1; i <= V-1; i++)
    {
        for (int j = 0; j < E; j++)
        {
```

```
            int u = g[j].second.first, v = g[j].second.second;
            int weight = g[j].first;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }
    for (int i = 0; i < E; i++)
    {
        int u = g[i].second.first, v = g[i].second.second;
        int weight = g[i].first;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            printf("Graph contains negative weight cycle");
    }
}
```

There is a nested two for loops. One for vertices and one for edges so the time complexity of this algorithm is $O(EV)$.

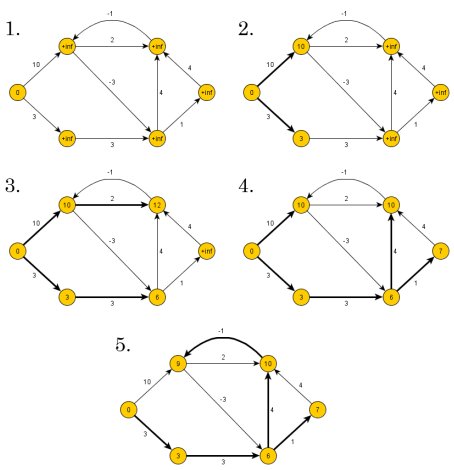

Figure 8: Example of Bellman-Ford Algorithm

Algorithm	Time Complexity	Space Complexity	Notes
Dijkstra Shortest Path A.	$O(ElogV)$	$O(V^2)$	Single Source Fails on negative edges
Floyd-Warshall A.	$O(V^3)$	$O(V^2)$	All Pairs Fails on negative cycle
Bellman-Ford A.	$O(VE)$	$O(V+E)$	Single Source Can detect negative cycle

As a result, we inspected three shortest path algorithm. Here is a brief conclusion as a table.