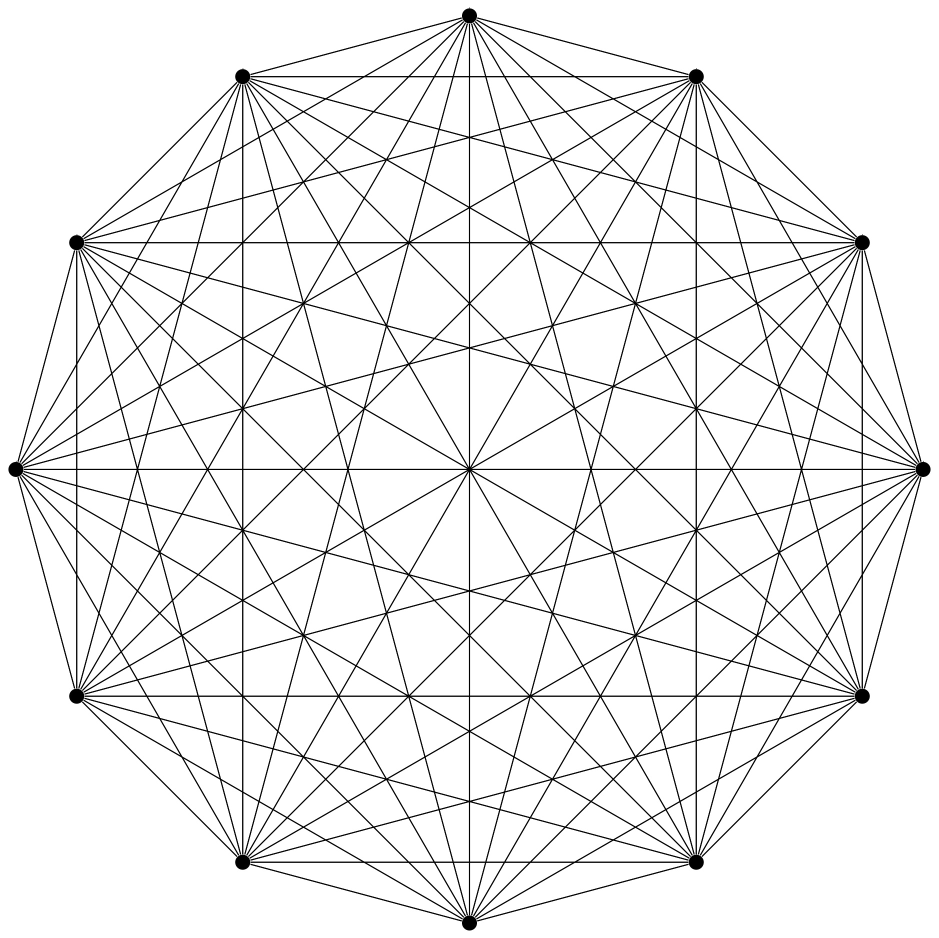
Graph Theory

Aniket Sadashiva
Mentor:Adwait Godbole

June 10, 2020



Contents

§0. Introduction

§§0.1. Basic Terminologies and Definitions

0.1.1 Graph

A graph G is an ordered pair, $G{=}(V(G), E(G))$ where $V(G)$ is a (finite) set of elements and $E(G)$ is a set of two subsets (pair of elements) of $V(G)$.

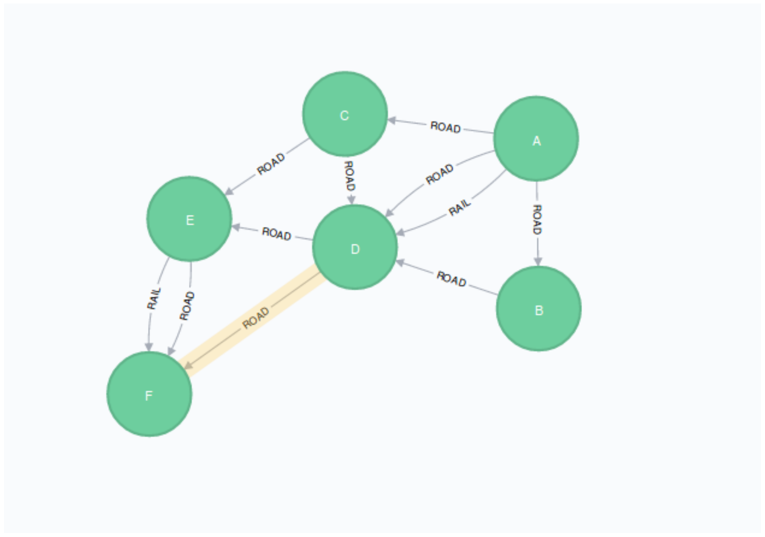


Figure 1: An Road Network

In the above figure the letters $A, B, .., F$ represent Vertices V and Roads represent Edges E.

0.1.2 Egde

An edge v,w is said to join the vertices v and w, and is usually abbreviated to vw .

0.1.3 Loop

An edge joining a vertex to itself.

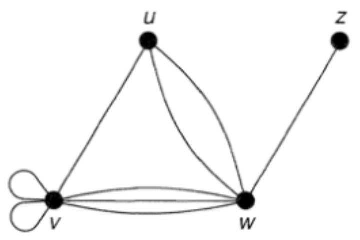


Figure 2: A Graph with loops

0.1.4 Simple Graph

A graph in which there is at most one edge between any pair of vertices and has no loops.

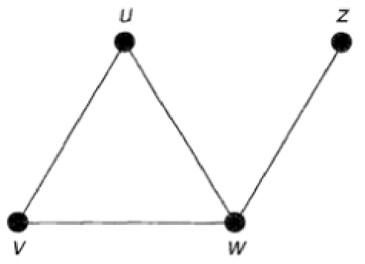


Figure 3: A Simple Graph

0.1.5 Directed Graph

A graph in which each edge has an associated direction ,indicated by an arrow.

0.1.6 Order , Degree and Size

Order is the number of vertices, **degree** is the number of edges associated to a vertex and **size** is the number of Edges in a graph.

0.1.7 Adjacent Edges

If two edges share a **common vertex** then they are said to be adjacent.

0.1.8 Complete Graph

A simple graph in which each pair of distinct vertices are adjacent is a complete graph. A Complete graph on n vertices is denoted by K_n.

Note: K_n has $\frac{n(n-1)}{2}$ edges.

0.1.9 Bipartite Graphs

A graph whose vertex set can be partitioned into two sets V_1 and V_2 such that for any edge uv $u \in V_1$ and $v \in V_2$.
A **Complete Bipartite Graph** has every possible edge between two sets of vertices.It is represented by $K_{n,m}$, where n and m are the number of elements in sets V_1 and V_2.

A **Star** is a graph with n = 1.

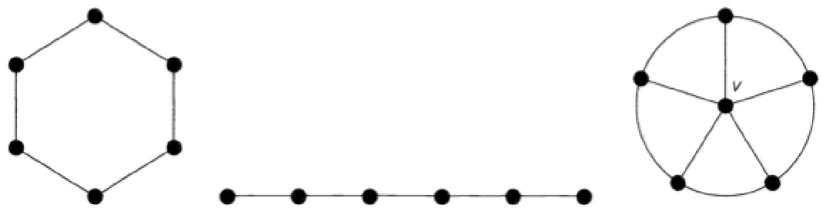
0.1.10 Cycles, Paths and Wheels

A connected graph that is regular of degree 2 is a **cycle**. We denote the cycle graph on n vertices by C_n. The graph obtained from C_n by removing an edge is the **path** on n vertices, denoted by P_n. The graph obtained from Cn-1 by joining each vertex to a new vertex v is the wheel on n vertices, denoted by W . The graphs C_6, P_6 and W_6 are shown below:

0.1.11 Regular Graphs

A graph in which each vertex has the same degree is a **regular graph** . If each vertex has degree r, the graph is regular of degree r or **r-regular**.

Note: A null graph N_n is regular of degree 0, the cycle graph C_n is regular of degree 2, and the complete graph K_n is regular of degree $n-1$.



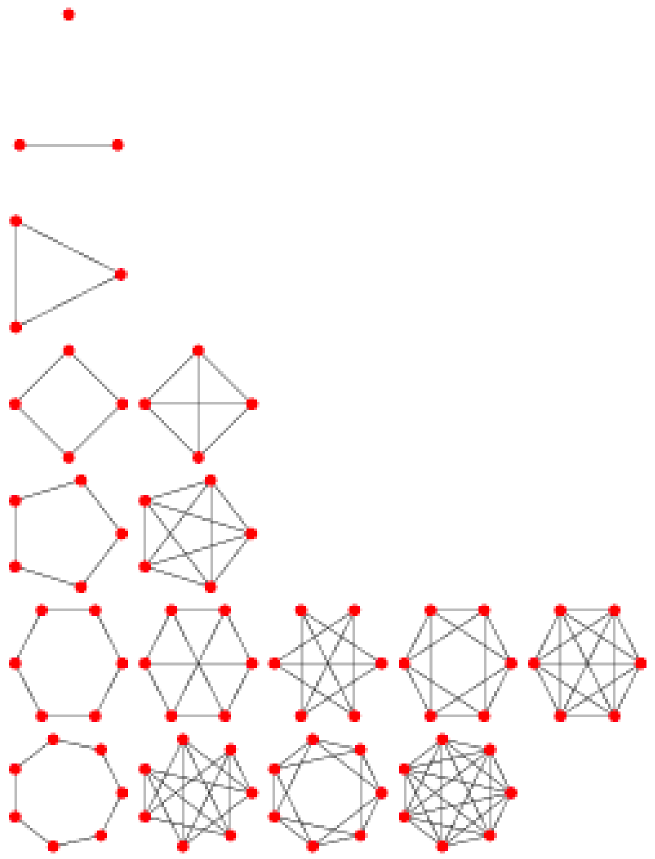


Figure 4: Regular Graphs

0.1.12 Subgraphs

A graph F is a subgraph of a graph G ,if $V(F) \subseteq V(G)$ and $E(F) \subseteq E(G)$.

§1. Representation of Graphs

§§1.1. Adjacency Matrix

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

1.1.1 Implementation :

- Make a 2D array of size VxV
- If graph is unweighted set adj[u][v]=1.
- If graph is weighted with weight w,set adj[u][v]=w.

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

Cons: Consumes more space $O(V^2)$. Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.

§§1.2. Adjacency List

An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.

1.2.1 Implementation :

```
void addEdge(vector<int> adjList[], int u, int v)
{
    adjList[u].push_back(v);
    adjList[v].push_back(u);
}
```

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

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

§2. Basic Theorems and Lemmas

§§2.1. Handshaking Lemma

If several people shake hands, then the total number of hands shaken must be even.
In terms of Graph Theory, this translates to : *In any graph the sum of all the vertex-degrees is an even number which is exactly twice the number of edges.*

Proof This happens precisely because just two hands are involved in each handshake.
Alternatively, we can also put it this way, each edge would be counted twice in each degree count for each of its endpoints.

Theorem 2.1. *If G is a bipartite graph, then each cycle of G has even length.*

Proof Since G is bipartite, we can split its vertex set into two disjoint sets A and B so that each edge of G joins a vertex of A and a vertex of B.Let $v_0 \to v_1 \to \cdot \to v_m \to v_0$ be a cycle in G, and assume (without loss of generality) that v_0 is in A. Then v_1 is in B, v_2 is in A, and so on. Since v_m must be in B, the cycle has even length.

Theorem 2.2. *Let G be a simple graph on n vertices. If G has k components, then the number m of edges of G satisfies*

$$n - k \leq m \leq \frac{(n-k)(n-k+1)}{2}$$

Proof For the lower bound , suppose that first component has n_1 vertices, second component has n_2 vertices $\cdots$ and so on such that $\sum_{i=1}^{k} n_i = n$.
Now each component has at least $n_i - 1$ edges and summing up for all components gives the desired lower bound.
For the upper bound we assume each component is a complete graph . Suppose, then, that there are two components C_i and C_j with n_i and n_j vertices, respectively, where n_i ¿ n_j ¿ 1. If we replace C_i and C_j by complete graphs on $n_i + 1$ and $n_j - 1$ vertices, then the total number of vertices remains unchanged, and the number of edges is changed by

$$n_i - n_j + 1$$

which is positive.It follows that, in order to attain the maximum number of edges, G must consist of a complete graph on $n - k + 1$ vertices and $k - 1$ isolated vertices. The result now follows.

Corollary *Any simple graph with n vertices and more than $\frac{(n-1)(n-2)}{2}$ edges is connected.*

Theorem 2.3. *If G is a graph in which the degree of each vertex is at least 2, then G contains a cycle.*

Proof If G has any loops or multiple edges, the result is trivial. We can therefore suppose that G is a simple graph. Let v be any vertex of G, We construct a walk $v_1 \to v_2 \to \cdot$ inductively by choosing v_1 to be any vertex adjacent to v and, for each i¿1, choosing v_{i+1} to be any vertex adjacent to v except v_{i-1}; the existence of such a vertex is guaranteed by our hypothesis. Since G has only finitely many vertices, we must eventually choose a vertex that has been chosen before. If v_k is the first such vertex, then that part of the walk lying between the two occurrences of v_k is the required cycle.

Theorem 2.4. *(Euler 1736). A connected graph G is Eulerian if and only if the degree of each vertex of G is even.*

Eulerian Graph A connected graph G is Eulerian if there exists a closed trail containing every edge of G.

Proof Suppose that P is an Eulerian trail of G. Whenever P passes through a vertex, there is a contribution of 2 towards the degree of that vertex. Since each edge occurs exactly once in P , each vertex must have even degree.
If the graph G contains loops or has multiple edges then the result follows.
Therefore we assume that G is simple.Since the degree of each vertex is even there is an entry and an exit ,so to say for each vertex hence we can get an Eulerian Graph.

Corollary 2.5. *A connected graph is Eulerian if and only if its set of edges can be split up into disjoint cycles.*

Corollary 2.6. *A connected graph is semi-Eulerian if and only if it has exactly two vertices of odd degree.*

Theorem 2.7. Let G be an Eulerian graph. Then the following construction is always possible, and produces an Eulerian trail of G.
Start at any vertex u and traverse the edges in an arbitrary manner, subject only to the following rules:
(i) erase the edges as they are traversed, and if any isolated vertices result, erase them too;
(ii) at each stage, use a bridge only if there is no alternative.

§3. Some important Results and Problems

§§3.1. Seven Bridges of Königsberg

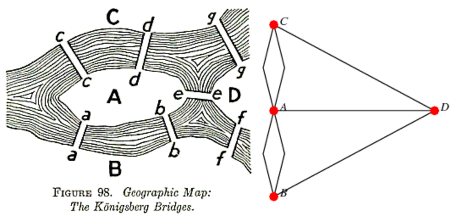


Figure 5: Königsberg Map

Problem Statement *The Königsberg bridge problem asks if the seven bridges of the city of Königsberg (left figure; Kraitchik 1942), formerly in Germany but now known as Kaliningrad and part of Russia, over the river Preger can all be traversed in a single trip without doubling back, with the additional requirement that the trip ends in the same place it began. This is equivalent to asking if the multigraph on four nodes and seven edges (right figure) has an Eulerian cycle. This problem was answered in the negative by Euler (1736), and represented the beginning of graph theory.*

Solution A vertex needs minimum of two edges to get in and out.If a vertex has odd edges then the person gets trapped.Hence every odd vertex should be a starting or ending point in the graph.
In our problem graph we have four odd vertices hence there cant be any Euler path possible.

§§3.2. Chinese Postman Problem

Problem Statement *Chinese postman problem, postman tour or route inspection problem is to find a shortest closed path or circuit that visits every edge of an undirected graph.*

Solution We can base our algorithm on the basis of following observations:
If all vertices are even, the total weight of the graph is the shortest distance based on theorems 4,5 and 6.
If there are only two odd vertices, pair up the arc/s between them, and then find the total weight of the graph.
If there are more than two odd vertices, pair up all of them and then find the minimum added weight.

§§3.3. Travelling Salesman Problem

Problem Statement *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?*

Solution It can be rephrased as the minimum cost Hamiltonian path problem. It turns out that this problem is NP-hard.Hence we opt for a brute-force solution.The plan is to traverse all cities in all possible orders i.e. generating all possible permutations of cities.This would give a solution with complexity $O(n!)$.However this could be improved by using dynamic programming and maintaining a tsp_matrix as shown in implementation.

Implementation I have used *mask* to keep record of cities travelled ,a *mask* 1010 implies cities B and D i.e. second and last cities have been travelled to.
Adjacency matrix *dist* is used to represent the cities and corresponding weights. The approach is straightforward traverse all required paths and then select the minimum of them.Here the complexity is of $O(n^2 2^n)$

Listing 1: Travelling Salesman Problem

```
int tsp(int mask,int pos){ //mask represents cities visited
                           //pos represents current city
    if(mask==visited_all)
        return dist[pos][0];
    if(tspmatrix[mask][pos]!=-1)
        return tspmatrix[mask][pos];
    int ans=INT_MAX;
    for(int city=0;city<n;city++){
        if(((mask)&((1<<city)))==0){        //city not visited
            int newAns=dist[pos][city] + tsp(mask|(1<<city), city);
            ans=min(ans,newAns);
        }


    }
return tspmatrix[mask][pos]=ans;
}
```

§§3.4. The 8 circles Problem

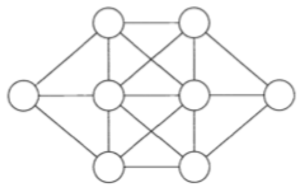


Figure 6: The 8 circles Problem

Problem Statement *Place the letters A, B, C, D, E, F, G, H into the eight circles in Fig. 4.1, in such a way that no letter is adjacent to a letter that is next to it in the alphabet.*

Solution Note that:
(i) the easiest letters to place are A and H, because each has only one letter to which it cannot be adjacent (namely, B and G, respectively)
(ii) the hardest circles to fill are those in the middle, as each is adjacent to six others.
This suggests that we place A and H in the middle circles. If we place A to the left of H, then the only positions of B and G are fixed and hence we can obtain a solution as shown:

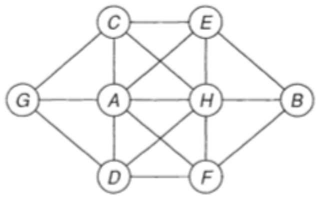


Figure 7: The 8 circles Problem Solution

§§3.5. Six people at a party

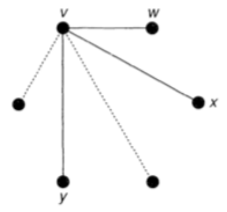


Figure 8: Six people at a party graphical representation

Problem Statement *Show that, in any gathering of six people, there are either three people who all know each other or three people none of whom knows either of the other two.*

Solution To solve this, we draw a graph in which we represent each person by a vertex and join two vertices by a solid edge if the corresponding people know each other, and by a dotted edge if not. We must show that there is always a solid triangle or a dotted triangle.

Let v be any vertex. Then there must be exactly five edges incident with v, either solid or dashed, and so at least three of these edges must be of the same type. Let us assume that there are three solid edges (see Fig.5 below) the case of at least three dashed edges is similar.

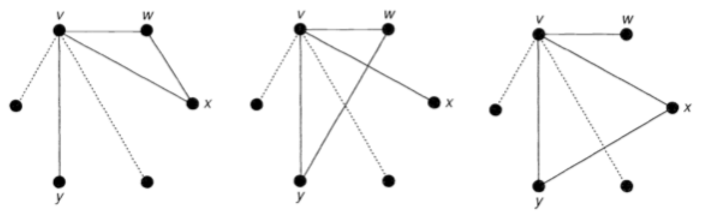


Figure 9:

If the people corresponding to the vertices w and x know each other, then v, w and x form a solid triangle, as required. Similarly, if the people corresponding to the vertices w and y, or to the vertices x and y, know each other, then we again obtain a solid triangle.
Finally, if no two of the people corresponding to the vertices w, x and v know each other, then w, x and y form a dotted triangle, as required.

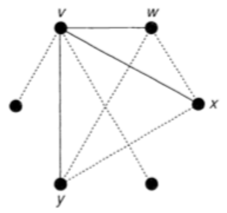


Figure 10:

§4. Graph Traversals

§§4.1. Breadth First Search

Breadth First Search or BFS is a traversing algorithm where we start traversing from a selected node (source or starting node) and traverse the graph level-wise thus exploring the neighbour nodes.We then move towards the next-level neighbour nodes.

4.1.1 Fire Spread Analogy :

The algorithm can be understood as a fire spreading on the graph: at the zeroth step only the source s is on fire. At each step, the fire burning at each vertex spreads to all of its neighbors. In one iteration of the algorithm, the "ring of fire" is expanded in width by one unit (hence the name of the algorithm).

4.1.2 Implementation :

To implement this algorithm, we need to use the Queue data structure. All the adjacent vertices are added into the queue, when all adjacent vertices are completed, one item is removed from the queue and start traversing through that vertex again.
The algorithm works in O(m+n) $O(m + n)$ time where n is the number of vertices and m is the number of edges.

```
void BFS(int s) {
q.push(s);
used[s] = true;
p[s] = -1;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int u : adj[v]) {
        if (!used[u]) {
            used[u] = true;
            q.push(u);
            d[u] = d[v] + 1;
            p[u] = v;
        }
    }
}
```

4.1.3 Applications of BFS :

- Find the shortest path from a source to other vertices in an unweighted graph.
- Finding the shortest cycle in a directed unweighted graph: Start a breadth-first search from each vertex. As soon as we try to go from the current vertex back to the source vertex, we have found the shortest cycle containing the source vertex. At this point we can stop the BFS, and start a new BFS from the next vertex. From all such cycles (at most one from each BFS) choose the shortest.
- Finding a solution to a problem or a game with the least number of moves, if each state of the game can be represented by a vertex of the graph, and the transitions from one state to the other are the edges of the graph.

§§4.2. Depth First Search

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from

the dead end towards the most recent node that is yet to be completely unexplored.
Depth First Search finds the lexicographical first path in the graph from a source vertex u to each vertex. Depth First Search will also find the shortest paths in a tree (because there only exists one simple path), but on general graphs this is not the case.

The algorithm works in $O(m+n)$ time where n is the number of vertices and m is the number of edges.

4.2.1 Implementation :

```
void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
}
```

4.2.2 Applications of DFS :

- Find any path in the graph from source vertex u to all vertices.
- Find lexicographical first path in the graph from source u to all vertices.
- Find the lowest common ancestor (LCA) of two vertices.

§5. Shortest Paths

§§5.1. Single Source Shortest Path Using BFS

We can modify our BFS algorithm to find the shortest path from the source to any node in an unwieghted graph(We have Dijkstra next for weighted graphs).The idea is that since BFS is a level order traversal ,it gives us the shortest distance.
We modify our visited array to distance array which will store the distance rather than boolean values. Additionally we initialize the $distance[source] = 0$ and all other nodes to infinity.

5.1.1 Implementation

```
void SSSP_BFS(T src){
    queue<T> q;
    map<T,int> dist;
    q.push(src);

    for(auto node_pair: l){
        T node =node_pair.first;
        dist[node]=INT_MAX;
    }

    dist[src]=0;
    while(!q.empty()){
        T node=q.front();
        q.pop();
        for(auto ngbr:l[src]){
            if(dist[ngbr]==INT_MAX){
                q.push(ngbr);
                //Modifying the distance at each level for each neighbour
                dist[ngbr]=dist[node]+1;
            }

        }
    }
}
```

§§5.2. Dijkstra's Algorithm

Problem *You are given a directed or undirected weighted graph with n vertices and m edges. The weights of all edges are non-negative. You are also given a starting vertex s.We have to find the lengths of the shortest paths from a starting vertex s to all other vertices, and output the shortest paths themselves.*
This problem is also called **Single-Source Shortest Paths Problem.**

5.2.1 Implementation :

We use a Greedy approach to solve this problem,
An array $d[v]$ is created where for each vertex v we store the current length of the shortest path from s to v in $d[v]$, and for all other vertices this length equals infinity. In the implementation a sufficiently large number (which is guaranteed to be greater than any possible path length) is chosen as infinity.

$$d[v] = \infty, v \neq s$$

In addition, we maintain a Boolean array $u[]$ which stores for each vertex v whether it's marked. Initially all vertices are unmarked:

$$u[v] = false$$

The Dijkstra's algorithm runs for n iterations. At each iteration a vertex v is chosen as unmarked vertex which has the least value $d[v]$: In the first iteration the starting vertex s will be selected.
The selected vertex v is marked. Next, from vertex v relaxations are performed: all edges of the form (v, to) are considered, and for each vertex to the algorithm tries to improve the value $d[to]$. If the length of the current edge equals len, the code for relaxation is:

$$d[\text{to}] = \min(d[\text{to}], d[v] + len)$$

After all such edges are considered, the current iteration ends. Finally, after n iterations, all vertices will be marked, and the algorithm terminates. We claim that the found values $d[v]$ are the lengths of shortest paths from s to all vertices v.

Code : The graph adj is stored as adjacency list: for each vertex v $adj[v]$ contains the list of edges going from this vertex, i.e. the list of pair¡int,int¿ where the first element in the pair is the vertex at the other end of the edge, and the second element is the edge weight.

The function takes the starting vertex s and two vectors that will be used as return values.

Listing 2: Dijkstra Algorithm

```
vector<vector<pair<int, int>>> Graph;

void Dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = Graph.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

        u[v] = true;
        for (auto edge : Graph[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}
```

§6. Disjoint Set Union

§§6.1. Motivation

Consider the following problem:

Problem 1. In a room are N persons, and we will define two persons are friends if they are directly or indirectly friends. If A is a friend with B, and B is a friend with C, then A is a friend of C too. A group of friends is a group of persons where any two persons in the group are friends. Given the list of persons that are directly friends find the number of groups of friends and the number of persons in each group.

Now this problem can be easily solved using BFS.However we use another data structure i.e. Disjoint Set Union to solve this problem.
As the name suggests Disjoint Set Union consists of union of sets which are pairwise disjoint i.e. any two sets have no common elements between them.Formally,

Definition 6.1. A **disjoint set union** is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

Definition 6.2. A **Superparent** (also called leader) can be used to check if two elements are part of the same set of not.It can be thought of as root of the graph that individual component of dsu represents.

This data structure provides the following capabilities. We are given several elements, each of which is a separate set. A *DSU* will have an operation to combine any two sets, and it will be able to tell in which set a specific element is.The following are the three major operations associated with it:

- **makeDSU(v) :** Creating a set consisting of new member v.
- **getSuperparent(v) :** To get the superparent of v.
- **makeUnion(a,b) :** To merge two specified sets a and b.

§§6.2. Implementation

6.2.1 makeDSU(v)

```
void makeDSU(int v) {
    parent[v] = v;
}
```

6.2.2 getSuperparent(v) (Naive Implementation)

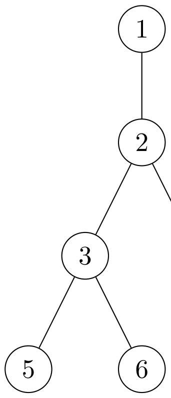
```
int getSuperparent(int v) {
    if (v == parent[v])
        return v;
    return getSuperparent(parent[v]);
}
```

6.2.3 makeUnion(a,b)

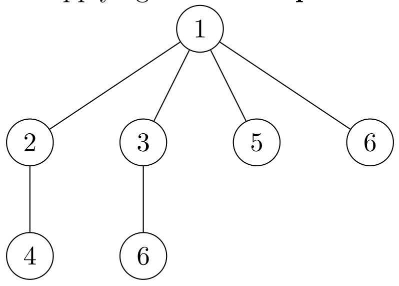
```
void makeUnion(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b)
        parent[b] = a;
}
```

§§6.3. Path Compression

We can see **getSuperparent** function is $O(v)$ in terms of time complexity.Using $PathCompression$ we can bring this to $O(1)$.
We will be making paths shorter by connecting the parents of node directly to the superparent of its parent as shown below for **getSuperparent(6) :**



After applying **Path Compression** :



```
int getSuperparent(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}
```

§§6.4. Union find or Cycle Detection ALgorithm

This can be used to check whether an undirected graph contains cycle or not. **Assumption:** The graph does not contain any self-loops.
The idea is to iterate over the pair of vertices given in disjoint components and if two of them have same superparent then there exists a cycle.

```
bool findCycle(Graph const &graph, int N)
{

        // consider every edge (u, v)
        for (int u = 1; u <= N; u++)
        {
                // Recur for all adjacent vertices
                for (int v : graph.adjList[u])
                {
                        // find root of the sets to which elements
                // u and v belongs
                int x = dsu.getSuperparent(u);
                int y = dsu.getSuperparent(v);

                // both u and v have same superparent, cycle is found
                if (x == y)
                        return true;
                else
                        dsu.Union(x, y);
            }
        }

        return false;
}
```

§7. Minimum Spanning Trees

§§7.1. Spanning Trees

Definition 7.1. A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices. Spanning trees prove important for several reasons:

1. They create a sparse subgraph that reflects a lot about the original graph.
2. They play an important role in designing efficient routing algorithms.
3. Some computationally hard problems, such as the Steiner tree problem and the traveling salesperson problem, can be solved approximately by using spanning trees.
4. They have wide applications in many areas, such as network design, bioinformatics, etc.

Theorem 7.1. For a complete graph ,K_n with n vertices the number of maximum possible spanning trees is n^{n-2}.

Definition 7.2. A **Prufer sequence** of length $n - 2$, for $n \leq 2$, is any sequence of integers between 1 and n, with repetitions allowed.

Lemma 7.2. There are n^{n-2} Prüfer sequences of length $n - 2$.

Proof : By definition, there are n ways to choose each element of a Prufer sequence of length $n2$. Since there are $n2$ elements to be determined, in total we have n^{n-2} ways to choose the whole sequence.

§§7.2. Minimum Spanning Tree

Definition 7.3. In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

There are two major algorithms,both are greedy algorithms that we'll be discussing about finding MSTs:

1. Kruskal's Algorithm
2. Prim's Algorithm

§§7.3. Kruskal's Algorithm

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.
This can be summarised in the following steps:

1. Make a forest F with each vertex as a seperate tree
2. create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning:
   (a) remove an edge with minimum weight from S
   (b) if the removed edge connects two different trees then add it to the forest F, combining two trees into a single tree

7.3.1 Implementation :

We'll use a disjoint set union data structure to implement Kruskal's Algorithm.

```
void Kruskals_algorithm(){
  Graph G;
  int i, u, v;
  sort(G.begin(), G.end());   // increasing weight
  for (i = 0; i < G.size(); i++) {
    u = G.getSuperparent(G[i].second.first);
    v = G.getSuperparent(G[i].second.second);
    if (u != v) {
      T.push_back(G[i]);   // add to tree
      G.makeUnion(u, v);
    }
  }
}
```

§§7.4. Prim's Algorithm

It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Vertices already included in MST.
- Vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.
It treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

7.4.1 Implementation :

```
// Function to find the vertex with minimum key value
int minKey_value(int key[], bool visited[])
{
    int min = 999, min_index;  // 999 represents an Infinite value

    for (int v = 0; v < V; v++) {
        if (visited[v] == false && key[v] < min) {
                // vertex should not be visited
            min = key[v];
                        min_index = v;
        }
    }
    return min_index;
}

// Function to print the final MST stored in parent[]
void printMST(int parent[], int cost[V][V])
{
    int minCost=0;
        cout<<"Edge \tWeight\n";
    for (int i = 1; i< V; i++) {
                cout<<parent[i]<<" - "<<i<<" \t"<<cost[i][parent[i]]<<" \n";
                minCost+=cost[i][parent[i]];
    }
```

```
        cout<<"Total cost is"<<minCost;

}

// Function to find the MST using adjacency cost matrix representation
void Prims_Algorithm(int cost[V][V])
{
    int parent[V], key[V];
    bool visited[V];

    // Initialize all the arrays
    for (int i = 0; i< V; i++) {
        key[i] = 999;    // 99 represents an Infinite value
        visited[i] = false;
        parent[i]=-1;
    }

    key[0] = 0;  // Include first vertex in MST by setting its key vaue to 0.
    parent[0] = -1; // First node is always root of MST

    // The MST will have maximum V-1 vertices
    for (int x = 0; x < V - 1; x++)
    {
        // Finding the minimum key vertex from the
        //set of vertices not yet included in MST
        int u = minKey_value(key, visited);

        visited[u] = true;  // Add the minimum key vertex to the MST

        // Update key and parent arrays
        for (int v = 0; v < V; v++)
        {
            // cost[u][v] is non zero only for adjacent vertices of u
            // visited[v] is false for vertices not yet included in MST
            // key[] gets updated only if cost[u][v] is smaller than key[v]
            if (cost[u][v]!=0 && visited[v] == false && cost[u][v] < key[v])
            {
                parent[v] = u;
                key[v] = cost[u][v];
            }
        }
    }

    // print the final MST
        printMST(parent, cost);
}
```

§§7.5. Minimum Spanning Tree vs. Traveling Salesman problem

A minimum spanning tree helps you build a tree which connects all nodes, or as in the case above, all the places/cities with minimum total weight.
Whereas, a traveling salesman problem (TSP) requires you to visit all the places while coming back to your starting node with minimum total weight.
Following are some of the other real-life applications of Kruskal's algorithm:

1. Landing Cables
2. TV Network
3. Tour Operations

§8. Network Flow

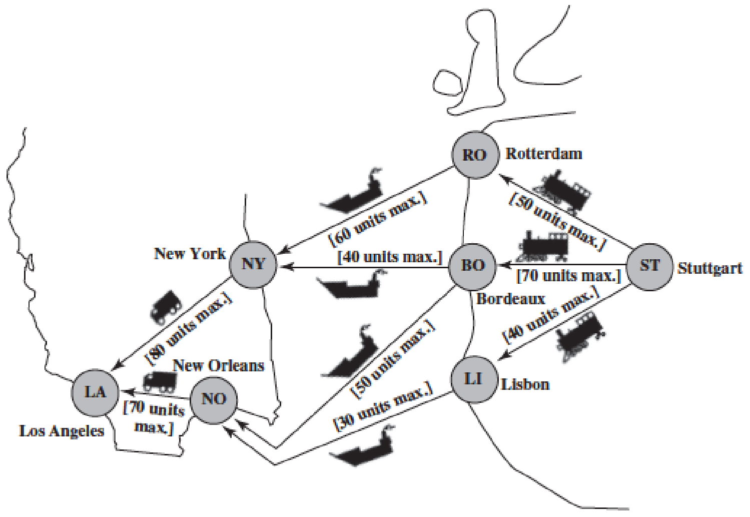


Figure 11: Maximal flow in the above Network

Definition 8.1. A **network** is a directed graph G with vertices V and edges E combined with a function c, which assigns each edge $e \in E$ a non-negative integer value, the capacity of e.

Definition 8.2. A **flow network** is a network with two special vertices **source** and **sink**.

Definition 8.3. A **flow** in a flow network is function f, that again assigns each edge e a non-negative integer value, namely the flow. The function has to fulfill the following two conditions:

- The flow of an edge cannot exceed the capacity.

$$f(e) \leq c(e)$$

- And the sum of the incoming flow of a vertex u has to be equal to the sum of the outgoing flow of u except in the source and sink vertices:

$$\sum_{(v,u)\in E} f((v,u)) = \sum_{(u,v)\in E} f((u,v))$$

A good analogy for a flow network is the following visualization: We represent edges as water pipes, the capacity of an edge is the maximal amount of water that can flow through the pipe per second, and the flow of an edge is the amount of water that currently flows though the pipe per second. This motivates the first flow condition. There cannot flow more water through a pipe than its capacity. The vertices act as junctions, where water comes out of some pipes, and distributes it in some way to other pipes. This also motivates the second flow condition. In each junction all the incoming water has to be distributed to the other pipes. It cannot magically disappear or appear. The source s is origin of all the water, and the water can only drain in the sink t.
The following image show a flow network. The first value of each edge represents the flow, which is initially 0, and the second value represents the capacity.

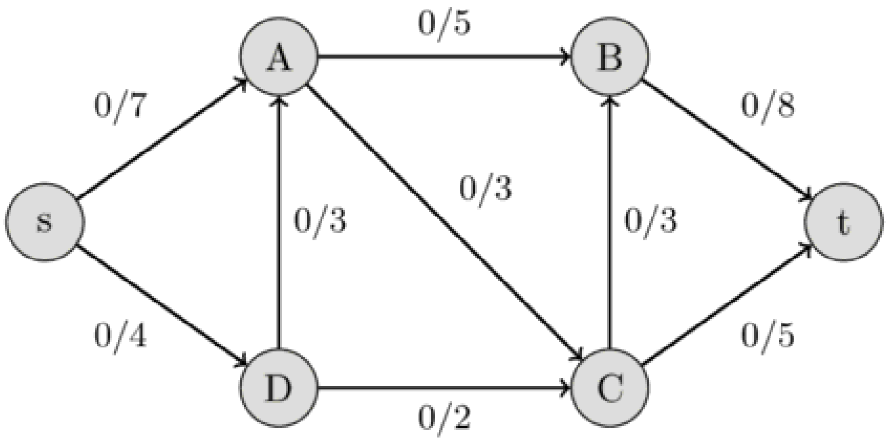


Figure 12: An Flow Network

The value of a flow of a network is the sum of all flows that gets produced in source s, or equivalently of the flows that are consumed in the sink t.A maximal flow is a flow with the maximal possible value. Finding this maximal flow of a flow network is the problem that we want to solve.
In the visualization with water pipes, the problem can be formulated in the following way: how much water can we push through the pipes from the source to the sink.

The following image show the maximal flow in the flow network.

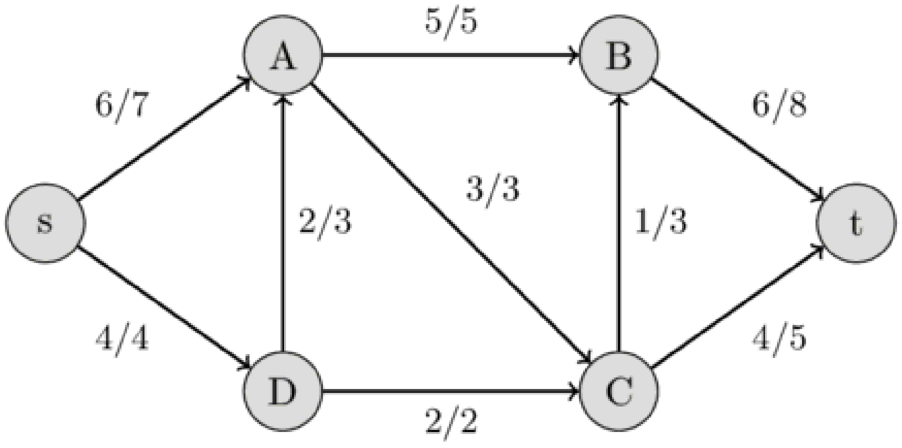


Figure 13: Maximal flow in the above Network

§§8.1. Motivation : The Max-flow Min-cut Problem

Given a network with vertices and edges between those vertices that have certain weights, how much "flow" can the network process at a time?

§§8.2. Ford-Fulkerson's Method :

It was discovered in 1956 by Ford and Fulkerson. This is referred to as a method because parts of its protocol are not fully specified and can vary from implementation to implementation. An algorithm typically refers to a specific protocol for solving a problem, whereas a method is a more general approach to a problem.

Definition 8.4. An **Augmented Path** is a path from source to sink through which flow can be sent.

8.2.1 Implementation :

The idea is to iterate over all augmented paths and send over them till none are left.

```
flow = 0
for each edge (u, v) in G:
    flow(u, v) = 0
while there is a path, p, from s -> t in residual network Graph:
    residual_capacity(p) = min(residual_capacity(u, v) : for (u, v) in p)
    flow = flow + residual_capacity(p)
    for each edge (u, v) in p:
        if (u, v) is a forward edge:
            flow(u, v) = flow(u, v) + residual_capacity(p)
        else:
            flow(u, v) = flow(u, v) − residual_capacity(p)
return flow
```

8.2.2 Complexity Analysis :

The analysis of Ford-Fulkerson depends heavily on how the augmenting paths are found. The typical method is to use BFS to find the path. If this method is used, Ford-Fulkerson runs in polynomial time.
Finding the augmenting path inside the while loop takes $O(V+E)$, where E is the set of edges in the residual graph. This can be simplified to $O(E)$. So, the runtime of Ford-Fulkerson is

$$O(E * |F|)$$

Where F is the maximum flow.

§§8.3. Edmond Karp Algorithm :

Finding the maximum flow for a network was first solved by the Ford-Fulkerson algorithm. A network is often abstractly defined as a graph, G, that has a set of vertices, V, connected by a set of edges, E. There is a source, s, and a sink, t, which represent where the flow is coming from and where it is going to. Finding the maximum flow through a network was solved via the max-flow min-cut theorem, which then was used to create the Ford-Fulkerson algorithm.

Edmonds-Karp is identical to Ford-Fulkerson except for one very important trait. The search order of augmenting paths is well defined. As a refresher from the Ford-Fulkerson wiki, augmenting paths, along with residual graphs, are the two important concepts to understand when finding the max flow of a network.

Augmenting paths are simply any path from the source to the sink that can currently take more flow. Over the course of the algorithm, flow is monotonically increased. So, there are times when a path from the source to the sink can take on more flow, and that is an augmenting path.

8.3.1 Implementation :

The idea is to iterate over all augmented paths and send over them till none are left.

```
#E=Adjacency List
#C=Capacity Matrix
#s=Source
#t=sink
def EdmondsKarp(E, C, s, t):
    n = len(C)
    flow = 0
    F = [[0 for y in range(n)] for x in range(n)]
    while True:
        P = [-1 for x in range(n)]
        P[s] = -2
        M = [0 for x in range(n)]
        M[s] = decimal.Decimal('Infinity')
        BFSq = []
        BFSq.append(s)
        pathFlow, P = BFSEK(E, C, s, t, F, P, M, BFSq)
        if pathFlow == 0:
            break
        flow = flow + pathFlow
        v = t
        while v != s:
            u = P[v]
            F[u][v] = F[u][v] + pathFlow
            F[v][u] = F[v][u] - pathFlow
            v = u
    return flow

def BFSEK(E, C, s, t, F, P, M, BFSq):
    while (len(BFSq) > 0):
        u = BFSq.pop(0)
        for v in E[u]:
            if C[u][v] - F[u][v] > 0 and P[v] == -1:
                P[v] = u
                M[v] = min(M[u], C[u][v] - F[u][v])
                if v != t:
                    BFSq.append(v)
                else:
                    return M[t], P
    return 0, P
```

References

[1] Robin J. Wilson, *Introduction to Graph Theory*, Addison Wesley Longman Limited,Essex. 4th edition, 1996.

[2] Gary Chartrand and Ping Zhang, *A First Course in Graph Theory*, McGraw-Hill,Boston. 1st edition, 2012.