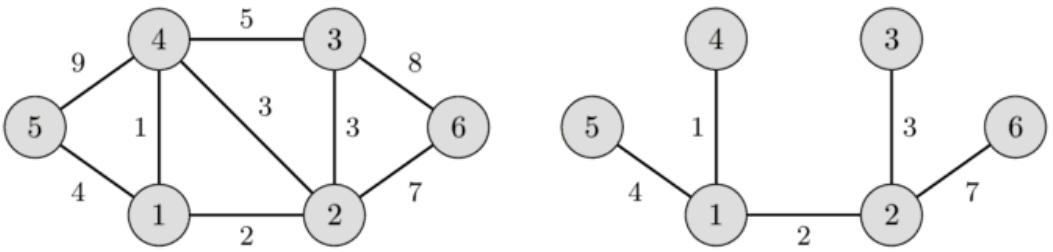
ALGORITHMS ANALYSIS AND DESIGN PROJECT

MINIMUM SPANNING TREE

ABSTRACT: This part of the project is an analytical comparison between two algorithms which are commonly used to solve minimum spanning tree (MST) problems. Those two algorithms are none other than **Kruskal** and **Prim.**
Kruskal is a greedy algorithm that works by sorting all the edges of the graph in non-decreasing order of the graph weight. Next step is to pick the smallest edge and check if it forms a cycle with the spanning tree that is already arranged so far. If the cycle is not formed then include this edge, otherwise leave it.
Prim algorithm (also known as Jarnik's algorithm) is another greedy algorithm which works by joining two subsets, one containing vertices already included in the MST, the other set contain the rest of the vertices which are not included, and we try to connect both subsets with minimum weight edge, hence the minimum spanning tree.
To help simulate the theories explained, the program for each method is written with c++ programming language. These programs show when different vertices and edges are inputted, the run time will also be different. This proves that the time complexity for each algorithm is based on the number of vertices and edges of the graph. The time complexity for Kruskal and Prim algorithms are both O(ElogV).

INTRODUCTION:
A spanning tree is a set of edges such that any vertex can reach any other by exactly one simple path. The spanning tree with the least weight is called a minimum spanning tree.



In this above example, we have 6 vertices and 9 edges on the left side. In the left image you can see a weighted undirected graph, and in the right image you can see the corresponding minimum spanning tree. We can observe by different examples that a minimum spanning tree contains necessarily n-1 edges.

PROBLEM:
We will analyze two algorithms which one is better between Kruskal's and

Prim's algorithm to solve the MST (Minimum Spanning Tree) problem, and we will use C++ as the programming language to solve the problem.

ALGORITHM:

PRIM'S ALGORITHM:

History:~ The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959. Therefore, it is also sometimes called the Jarník's algorithm, Prim–Jarník algorithm, Prim–Dijkstra algorithm or the DJP algorithm.

Description:~ The minimum spanning tree is built gradually by adding edges one at a time . At first the MST consists of a single vertex . Then the min weight edge outgoing from this vertex is selected and added to the spanning tree. Now the MST has two vertices and one edge. Now select and add the edge whose one end is one of the earlier verted but avoid making a cycle because that will increase the no. of edged which is not necessary. And so on, i.e. every time we select and add the edge with minimal weight that connects one selected vertex with one unselected vertex. The process is repeated until the spanning tree contains all vertices (or equivalently until we have n-1 edges). In the last if the tree is not connected then we can not make a minimum spanning tree from the given graph.

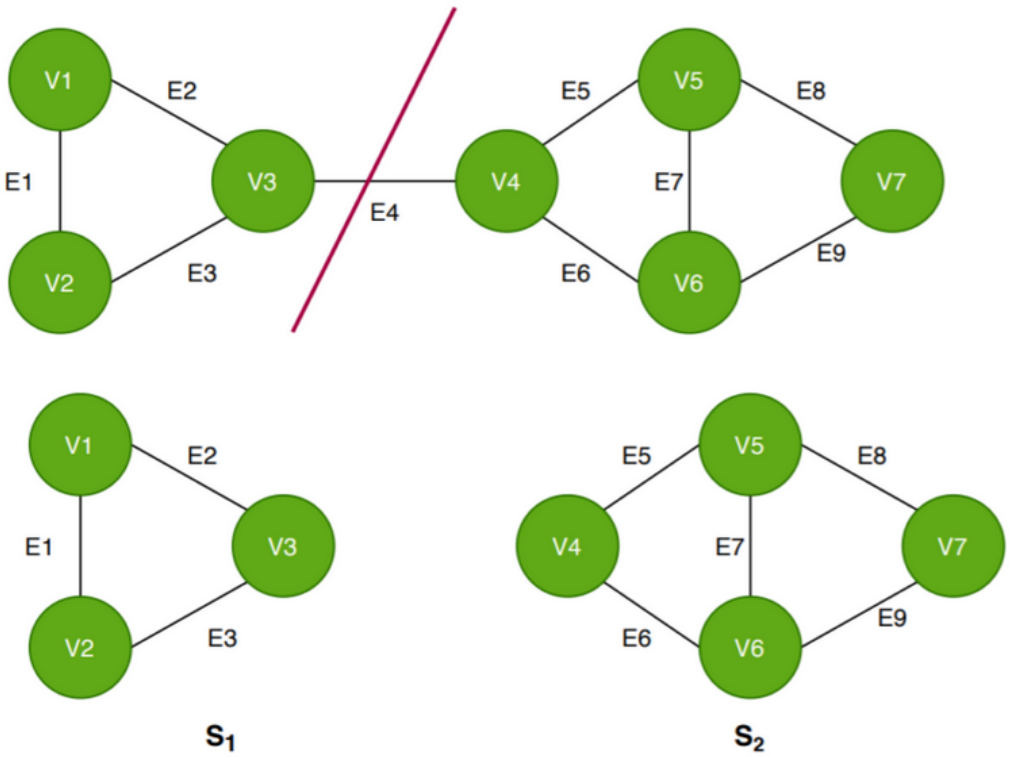
Proof:~

Cut property: In graph theory, a cut can be defined as a partition that divides a graph into two disjoint subsets. Let's define a cut formally. A cut C = (S1,S2) in a connected graph G(V,E) , partitions the vertex V into two disjoint subsets S1 and S2. We will define some terms regarding cut property like cut set, cut vertex and cut edge.

A cut set of a cut C(S1,S2) of connected graph G(V,E) can be defined as the set of edges that have one endpoint in S1 and the other in S2. For example, the C(S1,S2) of G(V,E) = {(i,j) ∈ $E | j \in S1, j \in S2$}

A vertex Vc is a cut vertex where there exists a connected graph G(V,E) and removing Vc from G disconnects the graph.

An edge Ec is cut edge of a connected graph G(V,E) if Ec ∈ E and G-Ec disconnects the graph.



So here the cut C disconnects the graph G and divides it into two components S1 and S2. In the figure V3 and V3 are two cut vertices because if we remove any of them that will disconnect the graph.
E4 is cut edge because if we remove edge E4 it will break the graph G into two subgraphs.
A cut set is defined as a set of edges whose two end points are in two graphs. Here a cut set of the cut C(S1,S2) on G would be {E4}.

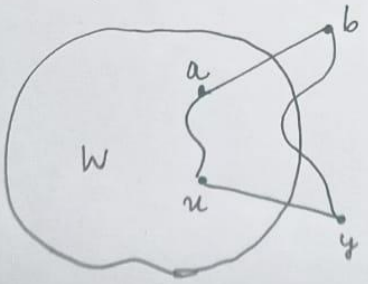
According to the cut property, if there is an edge in the cut set which has the smallest edge weight or cost among all other edges in the cut set, the edge should be included in the minimum spanning tree.

Proof of correctness of Prim's algorithm:
Theorem : If S is the spanning tree selected by Prim's algorithm for input graph G = (V, E), then S is a minimum-weight spanning tree for G

The proof is by contradiction, so assume that S is not minimum weight. Let Edge set for MST = $\{e_1, e_2, e_3 \dots e_{n-1}\}$, Let ~~be~~ U be a min-weight spanning tree that contains edges from the longest possible prefix of edge set.

Let $e_i = \{u, y\}$ be the first edge added to S by Prim's algo. that is not in U and Let W be the set of vertices immediately before $\{x, y\}$ is selected. Notice that it follows that U contain edges $e_1, e_2 \dots e_{i-1}$ but not edge e_i.

There must be a path $u \rightsquigarrow y$ in U, so let $\{a, b\}$ be the first edge on this path with one endpoint (a) inside W, and the another endpoint (b) outside W, as in the diagram



Define the set of edges $T = U + \{\{u, v\}\} - \{\{a, b\}\}$, and notice that T is spanning tree for graph G. Consider the three possible cases for the weights of edges $\{u, y\}$ and $\{a, b\}$.

Case 1, $w(\{a,b\}) > w(\{u,y\})$. In this case, in creating T we have added an edge that has smaller weight then the one we removed, and so $w(T) < w(U)$. However, this is impossible, since U is min. spanning Tree.

Case 2, $w(\{a,b\}) = w(\{u,y\})$. In this case $w(T) = w(U)$, so T is also a min. spanning tree. Since Prim's algo hasn't selected edge $\{a,b\}$ yet that edge can not be one of $e_1, e_2, e_3 \ldots e_{i-1}$. This implies that T contain edges $e_1, e_2, \ldots e_i$ which is longer prefix of E_s than U contains. This contradicts the definition of tree U.

Case 3, $w(\{a,b\}) < w(\{u,y\})$. In this case, since the weight of edge is smaller. Prim's algorithm will select $\{a,b\}$ at this step. The contradicts the definition of edge $\{u,y\}$

Since all possible cases lead to contradicts, our original assumption (that S is not minimum weight) must be invalid. This proves the theorem.

IMPLEMENTATION:~

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning trees, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked. Using adjacency list:

```
long long prim(int x)
{
    priority_queue<pii, vector<pii>, greater<pii>> Q;
    int y;
    long long minimumCost = 0;
    pii p;
    Q.push(make_pair(0, x));
    while (!Q.empty())
    {
        p = Q.top(); // Select the edge with minimum weight
        Q.pop();
        x = p.second; // Checking for cycle
        if (marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
        for (int i = 0; i < adj[x].size(); ++i)
        {
            y = adj[x][i].second;
            if (marked[y] == false)
                Q.push(adj[x][i]);
        }
    }
    return minimumCost;
}
```

Running Time of Prim's Algorithm:~

Let's assume that we are given **V** vertices and **E** edges in a graph for which we need to find an MST.

- ➔ To complete one iteration, we delete the minimum node from Min-Heap and add some no.of edge weights to Min-Heap.
- ➔ In total we delete **V** nodes from Min-Heap since we have **V** nodes in the graph and in every iteration **1** edge and in total **V-1** edges in MST are deleted and each deletion takes complexity of **O(log(V))**. And we add at most **E** edges altogether where each addition takes complexity of **O(log(V))**.
- ➔ So, total complexity is **O((V+E)Log(V))**.

Best and Worst Cases for Prim's:~

- ➔ Best case time complexity of Prim's is when the given graph is a tree itself and each node has a minimum number of adjacent nodes.
- ➔ Worst case time complexity would be when it is a graph with $\mathbf{V^2}$ edges.

Space Complexity of Prim's:~

- ➔ We need an array to know if a node is in MST or not. Space **O(V)**.
- ➔ We need an array to maintain Min-Heap. Space **O(E)**.
- ➔ So, Total space complexity is of order **O(V+E)**.

KRUSKAL'S ALGORITHM:

History:~ This algorithm was described by Joseph Bernard Kruskal, Jr. in 1956.
Description:~ Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows a greedy approach as in each iteration it finds an edge which has least weight and adds it to the growing spanning tree. Before the execution of the algorithm, all edges are sorted by weight (in non-decreasing order). Then begins the process of unification: pick all edges from the first to the last (in sorted order), and if the ends of the currently picked edge belong to different subtrees, these subtrees are combined, and the edge is added to the answer. After iterating through all the edges, all the vertices will belong to the same sub-tree, and we will get the answer.
Proof:~ Let T be a spanning tree and we will prove this using induction. Let T' be a min weight spanning tree. If T = T', then T is a min weight spanning tree. If $T \neq T'$ then there exists an edge $e \in T'$ of minimum weight that is not in T. Further , T U e contains a cycle C such that :
→ Every edge in C has weight less than wt(e).
→ There is some edge f in C that is not in T' because T' does not contain the cycle C.

Consider the Tree T2 = T\{e} U {f}:
→ T2 is a spanning tree.
→ T2 has more edges in common with T' then T did.
→ And wt(T2) ≥ wt(T) , exchange an edge for one that is no more expensive.
We can redo the same process with T2 to find a spanning tree T3 with more edges in common with T' . By induction , we can continue this process until we reach T' , from which we see
wt(T) ≤ wt(T2) ≤ wt(T3)≤ ……..≤wt(T')
Since T' is a minimum weight spanning tree then these inequalities must be equalities and we conclude that T is a minimum weight spanning tree.

Implementation:~
Algorithm Steps:

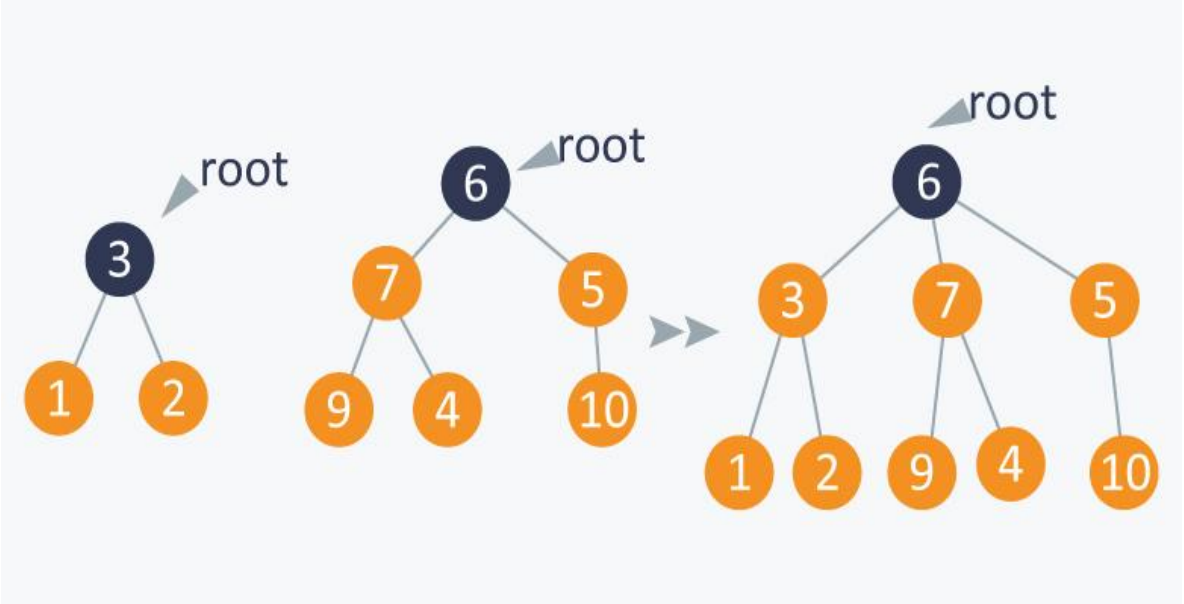
- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which don't form a cycle , edges which connect only disconnected components.

So now the question is whether the two vertices are connected or not.This could be done using DFS which starts from the first vertex, then checks if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is "Disjoint Sets"**.** Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

We will show Disjoint set example :

If you want to connect 1 and 5, then connect the root of subset A (the subset that contains 1) to the root of subset B (the subset that contains 5) because subset A

contains less number of elements than subset B.



In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

```cpp
int id[MAX], nodes, edges;
pair<ll, pair<int, int>> p[MAX];

void initialize() //making parent of their own
{
    for (int i = 0; i < MAX; ++i)
        id[i] = i;
}

int root(int x)  //finding the parent or root
{
    while (id[x] != x)
    {
        id[x] = id[id[x]];
        x = id[x];
    }
    return x;
}

void union1(int x, int y)   //joining two separate subsets
{
    int a = root(x);
    int b = root(y);
    id[a] = id[b];
}
```

```cpp
ll kruskal(pair<ll, pair<int, int>> p[])
{
    int x, y;
    ll cost, minimumCost = 0;
    for (int i = 0; i < edges; ++i)
    {

        x = p[i].second.first;
        y = p[i].second.second;

        cost = p[i].first;

        if (root(x) != root(y))
        {
            minimumCost += cost;
            union1(x, y);
        }
    }
    return minimumCost;
}
```

Running Time of Kruskal's:~

Let's assume that we are finding MST of a **N** vertices graph using Kruskal's.

- To check edges we need to sort the given edges based on weights of edges. The best way to sort has an order of **O(N log(N))**.
- To Check one edge if it needs to be in MST or not, we apply Union-find to check if it forms a circle with edges present and add to MST **exactly once** and apply the Union-Find algorithm of order **log(E)**.
- Since we perform at most **N** checks for a graph, total complexity is **O(Nlog(E))** for the checkings.
- So, total complexity is **O(Nlog(E)+Nlog(N))**

Best and Worst Cases for Kruskal's

For regular Kruskal's, time complexity will be **O(Nlog(E)+Nlog(N))** in all cases. For Kruskal's :

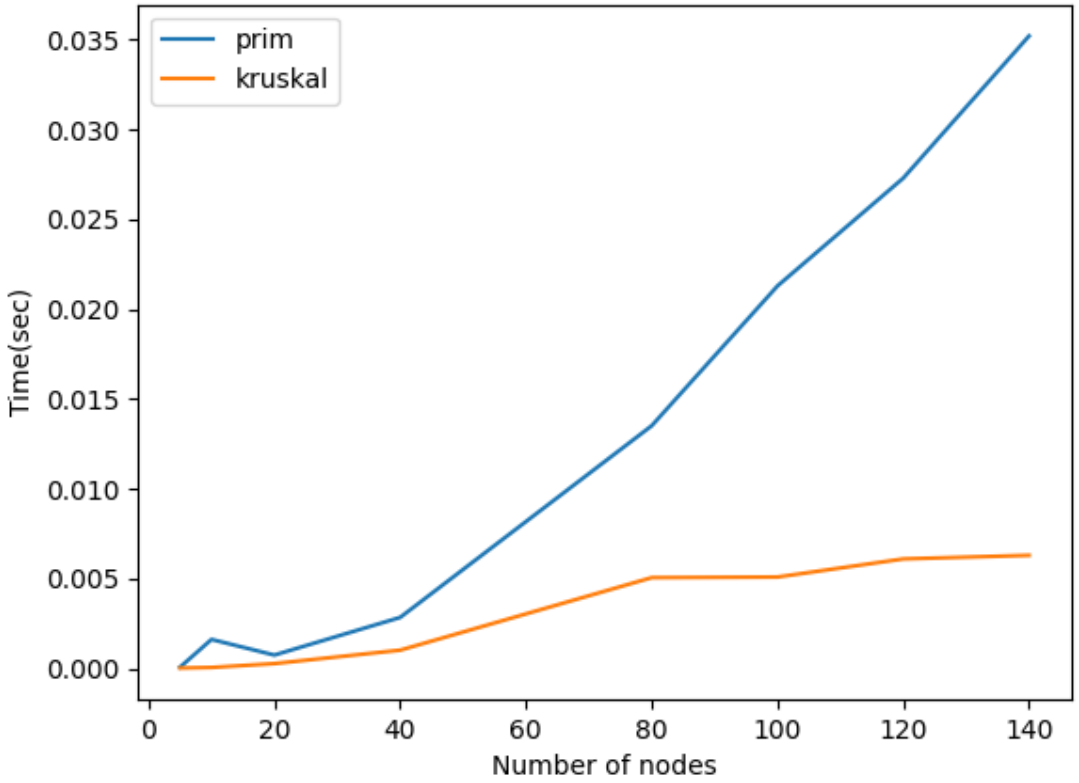
- In the best case scenario, we have **N** no cycles and we have to run **N-1** iterations to determine MST.
- Time complexity will be ***O((N-1)log(E)+Elog(E))** in this case.
- In the worst case we will have to check all E edges . Time complexity in such case would be **O(Elog(E)+Nlog(N))**

Try out the demo below and look out for the number of checkings performed for different types of graphs!

Space Complexity of Kruskal's:~

While sorting, we need an extra array to store the sorted array of edges (Space complexity of **O(E)**), Another array for Union-Find of size **O(E)**. So, total Space Complexity would be **O(log(E))**.

RUNNING TIME COMPARISON



This plot is made by using python matplotlib and c++ code in which user gives input the number of nodes and and there will be **nodes*(nodes+1)/2** whose weight is random. The code for the following algorithm comparison is given in the repository. This is a very varying plot only because of the number of edges as this is a large number because in code , there is an edge between every pair. This is the worst case plot as there are maximum possible edges.

MAX FLOW NETWORK

ABSTRACT: This part of the project is an analytical comparison among three algorithms which are commonly used to solve maximum-flow network problems. Those three algorithms are **Ford-Fulkerson** algorithm, **Denic's** algorithm and **Edmond-Karp's** algorithm.

INTRODUCTION: The maximum flow problem was first formulated in 1954 by T. E. Harris and F. S. Ross as a simplified model of Soviet railway traffic flow. The general problem can be described as: given a network, as well as a source (start) and a sink (destination) in that network, how does one route as much flow as possible from the source to the sink?
General characteristics of the networks where the problem is applicable are:

- Source: *"materials"* are produced at a steady rate
- Sink: *"materials"* are consumed at the same rate
- Flows through conduits are constrained to max values

So a number of tries have been made till now on this problem and all have some different approaches , We will discuss only few algorithms .

Year	Authors	Running Time
1956	Ford, Fulkerson	$O(nmU)$
1970	Dinic	$O(n^2m)$
1972	Edmonds, Karp	$O(nm^2)$
1974	Karzanov	$O(n^3)$
1977	Cherkasky	$O(n^2\sqrt{m})$
1978	Malhotra, Kumar, Maheshwari	$O(n^3)$
1979	Gali, Naamad	$O(nm\log^2 n)$
1980	Gali	$O(n^{\frac{5}{3}}m^{\frac{2}{3}})$
1983	Sleator, Tarjan	$O(nm\log n)$
1984	Tarjan	$O(n^3)$
1985	Gabow	$O(nm\log U)$
1988	Goldberg, Tarjan	$O(nm\log\frac{n^2}{m})$
1989	Auija, Orlin	$O(nm+n^2\log U)$
1989	Auija, Orlin, Tarjan	$O(nm\log\left(\frac{n}{m}\sqrt{\log U}+2\right))$
1989	Cheriyan, Hagerup	$E\left(\min\begin{pmatrix} nm\log n \\ nm+n^2\log^2 n\end{pmatrix}\right)$
1990	Alon	$O(\min\{nm\log n, n^{\frac{8}{3}}\log n\})$
1992	King, Rao	$O(nm+n^{2+\epsilon})$
1994	King, Rao, Tarjan	$O(nm\log_{\frac{m}{n\log n}} n)$
1998	Goldberg, Rao	$O(\min\{n^{\frac{2}{3}},\sqrt{m}\}m\log(\frac{n^2}{m})\log U)$
2012	Orlin	$O(nm+m^{31/16}\log^2 n)$

In 1955, Lester R. Ford. Jr and Delbert R. Fulkerson created the first known algorithm, the Ford-Fulkerson algorithm. Over the years, various improved solutions to the maximum flow problem were discovered,
notably the shortest augmenting path algorithm of Edmonds and Karp and independently Dinitz; the blocking flow algorithm of Dinitz; the push-relabel algorithm of Goldberg and Tarjan(not in project).
To help simulate the theories explained, the program for each method is written with c++ programming language. These programs show when different vertices and edges are inputted, the run time will also be different.
A network is a directed graph G with vertices V and edges E combined with a function c , which assigns each edge e ∈ E a non negative integer value , the capacity of e. Such a network is called a flow network.
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.

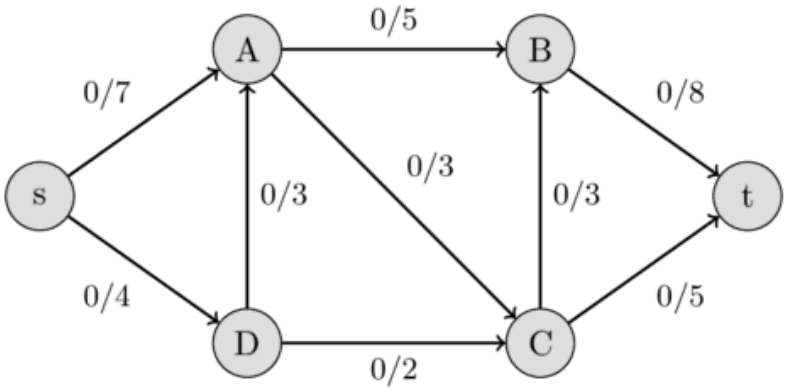
- The flow of an edge can not exceed the capacity.

  f(e) ≤ $c(e)$

- 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.

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

The below image shows a flow network , the first value of each edge represents the flow , which is initially 0 and the second value represents the capacity.



PROBLEM:

The value of 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. We have to solve the problem of finding the maximum flow of a flow network.

ALGORITHM:

FORD-FULKERSON ALGO

History:~ It was developed by L. R. Ford, Jr. and D. R. Fulkerson in 1956. A pseudocode for this algorithm is written below .

Description:~ We define residual capacity of a directed edge as the capacity minus the flow. It should be noted that if there is flow along some directed edge (u,v) then the reversed edge has capacity 0 and we can define the flow of it as f((u,v)) = -f((v,u)). This also defined the residual capacity for all reversed edges.

Initialise flow in all edges to 0
While(there exists an augmenting path(P) between S and T in residual network graph):
    Augment flow between S to T along the path P
    Update residual network graph
Return

Augmenting Path: An augmenting path is a simple path from source to sink which does not include any cycles and that passes only through positive weighted edges. A residual network graph indicates how much more flow is allowed in each edge in the

network graph. If there are no augmenting paths possible from Source to Sink , then the flow is maximum. The result i.e. the maximum flow will be the total flow out of the source node which is also equal to total flow into the sink node.

IMPLEMENTATION:

- An augmenting path in the residual graph can be found using DFS or BFS.
- Updating residual graph includes following steps:
  - ➢ For every edge in the augmenting path, a value of minimum capacity in the path is subtracted from all the edges of that path.
  - ➢ An edge of equal amount is added to edges in reverse direction for every successive node in the augmenting path.

It should be noted that the Fork-Fulkerson method doesn't specify a method of finding the augmenting path. We have possible approaches such as DFS or BFS which both work in O(E) .

Below implementation of Ford Fulkerson using BFS

```
int fordFulkerson(int graph[100][100], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph
    // with given capacities in the original graph as
    // residual capacities in residual graph
    int rGraph[V][V]; // Residual graph where rGraph[i][j]
                      // indicates residual capacity of edge
                      // from i to j (if there is an edge. If
                      // rGraph[i][j] is 0, then there is not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to
                   // store path

    int max_flow = 0; // There is no flow initially

    // Augment the flow while there is path from source to
    // sink
    while (bfs(s,t))
    {
        // Find minimum residual capacity of the edges along
        // the path filled by BFS. Or we can say find the
        // maximum flow through the path found.
        int path_flow = INT_MAX;
        for (v = t; v != s; v = parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }
```

```
        // update residual capacities of the edges and
        // reverse edges along the path
        for (v = t; v != s; v = parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        // Add path flow to overall flow
        max_flow += path_flow;
    }

    // Return the overall flow
    return max_flow;
}
```

MAX FLOW MIN CUT

The max-flow min-cut theorem stated that the maximum flow through any network from a given source to a given sink is exactly equal to the minimum sum of a cut . This theorem can be verified using the Fork-Fulkerson algorithm. This algorithm finds the maximum flow of a network or graph. Let's define the max min cut theorem formally. Let G = (X,A) be a network ϕ a flow on G. The value of the maximum flow from the source S to sink T is equal to the capacity of the minimum cut CT separating S and T.

$$Max(\phi) = Min(C(CT))$$

Complexity:~

- ➔ Worst case time complexity: O**(max_flow * E)**
- ➔ Average case time complexity: O**(max_flow * E)**
- ➔ Best case time complexity: O**(max_flow * E)**
- ➔ Space complexity: O**(V+ E)**

EDMONDS-KARP ALGORITHM

History:~

The Edmonds-Karp algorithm is one of the first and simplest max flow algorithms. It was published in 1972 by Jack Edmonds and Richard Karp.

Description:~

The algorithm works by repeatedly finding the shortest augmenting path using a breadth first search from s to t. When such a path P=(v1, v2,..., vk) where k ≥2, v1=s, vk=t is found, it calculates the bounding minimum capacity on that path, and sends that much flow over the same path. It keeps doing this in the residual network until no more augmenting paths exist. Correctness follows from the fact that the algorithm terminates when no more augmenting paths from s to t are found in the residual network, and the fact that the algorithm always keeps a valid flow. The algorithm never violates any capacity constraints, because when it sends flow, it sends flow according to the minimum residual capacity on the path. It also never produces any excess in nodes other than s and t, because all flow is pushed along paths from s to t.

The algorithm performs a breadth first search for each augmenting path in the graph. A single breadth first search takes O(m) time. Every time the algorithm finds an

augmenting path, it does a push along it. There must be at least one edge (u, v) on this path that is saturated, namely the edge with the minimum capacity. For this edge to be in the path, the distance from s to u must be less than the distance from s to v. After the edge has been saturated, it can not be used again before flow has been pushed the opposite way, which requires that the distance from s to v becomes less than the distance from s to u. The distance from s to any node can not be greater than n, and if the distance never decreases, so an edge can only be saturated n times. The only way we modify the distances is by pushing flow along the augmenting path. Saturated edges are effectively removed, and back edges are added back in if their residual capacity was zero. Removing an edge can not reduce the distance to a node. Adding an edge could, but the edges (vi, vi-1) we might add point the opposite way on the augmenting path which was found in a breadth first search. Adding (vi, vi-1) back in can not reduce the distance to vi-1, because the distance to vi was already greater than the distance to vi-1.

Implementation:~

The algorithm is identical to the Ford–Fulkerson algorithm, except that the search order when finding the augmenting path is defined. The path found must be the shortest path that has available capacity. This can be found by a **breadth-first search**, where we apply a weight of 1 to each edge. The running time of $O(VE^2)$ is found by showing that each augmenting path can be found in O(E) time, that every time at least one of the E edges becomes saturated (an edge which has the maximum possible flow), that the distance from the saturated edge to the source along the augmenting path must be longer than last time it was saturated, and that the length is at most V. Another property of this algorithm is that the length of the shortest augmenting path increases monotonically.

```
int capacity[100][100];
int maxflowEDMAND(int s, int t)
{
    int flow = 0;
    vector<int> parent(n);
    int new_flow;

    while (new_flow = bfs(s, t))
    {
        flow += new_flow;
        int cur = t;
        while (cur != s)
        {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }
    return flow;
}
```

Complexity:~

- ➔ Worst case time complexity: **O(VE^2)**
- ➔ Average case time complexity:**O(VE^2)**
- ➔ Best case time complexity: **O(VE^2)**
- ➔ Space complexity: **O(E + V)**

DINIC'S ALGORITHM

History:~

In 1970, Y. A. Dinitz developed a faster algorithm for calculating maximum flow over the networks. It includes construction of level graphs and residual graphs and finding of augmenting paths along with blocking flow.

Description:~

Residual Network: A residual network G^R of network G is a network which contain two edged for each edge (v,u) ∈ G.

- (v,u) with capacity $c^{R}_{vu} = c_{vu} - f_{vu}$
- (u,v) with capacity $c^{R}_{uv} = f_{vu}$

Blocking flow: A blocking flow of some network is such a flow that every path from s to t contains at least one edge which is saturated by this flow. It is not necessary that blocking flow is maximal.

Layered network: A layered network G is a network built in the following way. Firstly for each vertex v we calculate level[v] - the shortest path from s to this vertex using only edges with positive capacity. Then we keep only those edges (v,u) for which level[v]+1 = level[u], this network is acyclic.

The initial intention was just to accelerate the Ford-Fulkerson algorithm by means of a smart data structure. Note that finding an augmentation path takes O(m) time and becomes a bottleneck of the Ford-Fulkerson algorithm. If only a BFS tree was used, saturation of a bottleneck edge will disconnect s and t. Thus, it is invaluable to save all information gathered in BFS for subsequent iterations. For this aim, the BFS tree is enriched to a layered network: BFS tree: recording only the first edge found to a node v; Layered network: recording all the edges residing on shortest s −t paths in residual graph. Once layer numbers were calculated for nodes, a shortest s −t path could be found in O(n) time rather than O(m) time.

Shimon Even and Alon Itai understood the paper by Y. Dinitz except for the layered network maintenance and that by A. Karzanov. The gaps were spanned by using: Blocking flow (first proposed by A. Karzanov and implicit in the paper by Y. Dinitz): A blocking flow, also known as shortest saturation flow, aims to saturate all shortest s −t paths in a residual network. After augmenting with a blocking flow, the level number of node t increases by at least 1. DFS: Dinic's algorithm uses DFS technique to find the shortest path in a layered network. Only O(n) time is needed as it exploits level numbers of nodes. In contrast, the Edmonds-Karp algorithm uses BFS technique to find the shortest path in a residual graph, which needs O(m) time.

Algorithm:~

```
Initialize f(e) = 0 for all e.
   while TRUE  do
      Construct layered network Nf from residual graph Gf
      using extended BFS technique;
if t is unreachable from s in Gf then
      break;
end if
```

Find a blocking flow b_f in N_f using DFS technique guided

by the layered network;

Augment flow f = f + b_f;

end while

return f;

Construct-Layered-Network(G_f)

Set d_f(s) = 0, d_f(v) = ∞ for node v ≠ s, and add s into queue Q;

Set layered network N_f = (V_f,E_f) as V_f = {s} and E_f = {};

while Q is not empty do

v = Q.dequeue();

for each edge (v,w) in G_f do

if d_f(w) = ∞ then

Q.enqueue(w); d_f(w) = d_f(v) + 1;

V_f = V_f ∪ {w}; E_f = E_f ∪ {(v,w)};

end if

if d_f(w) = d_f(v) + 1 then

E_f = E_f ∪ {(v,w)};

end if

end for

end while

Perform BFS in N_f from t with all edges directions reversed, and delete v from N_f if v cannot be visited.

- The difference from the standard BFS procedure is that for any edge (v,w) with d_f(w) = d_f((v) + 1 will be added to N_f even if w has already been added to Q. Thus, for each vertex v, exactly all edges in shortest paths from s to v are added in N_f.
- The nodes (and their incident edges) not on the shortest paths from s to t will be removed from N_f, e.g., the node in dash.

Dinic-Blocking-Flow(Nf)

Set bf as 0-flow;

while there exists an edge from s in N_f do

Find a path p from s of maximal length in N_f;

if p leads to t then

b_f =augment(p,b_f);

Remove from N_f the bottleneck edges in p;

else

Delete the last node in p (and incident edges);

end if

end while

return b_f ;

Complexity:~

- ➔ Worst case time complexity: **O(E V^2)**
- ➔ Average case time complexity: **O(E V^2)**
- ➔ Best case time complexity: **O(E V^2)**
- ➔ Space complexity: **O(E + V)**

```
long long DinicFlow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
```

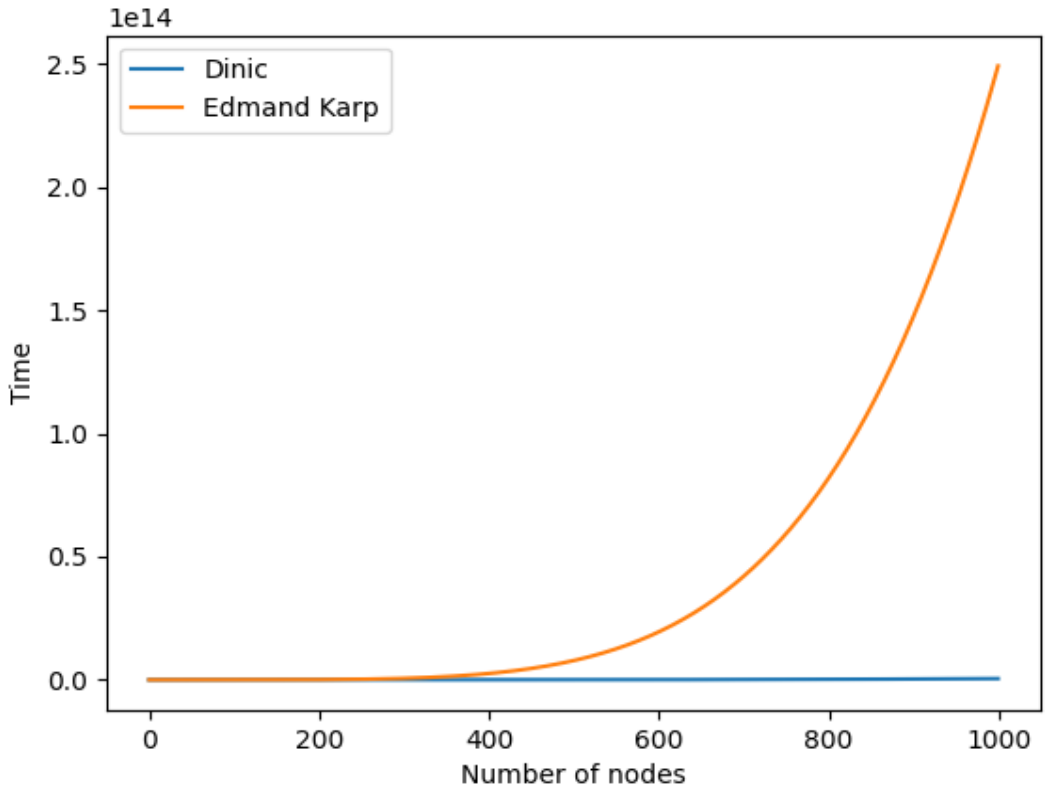
Conclusion

The first algorithm for solving the max flow problem was Ford Fulkerson . An algorithm that iteratively finds augmenting paths. Since they posed no restrictions on the order with which paths are found, their algorithm runs in O(nmU) for integer capacity constraints. This bound is achieved by observing that nU is an upper bound on the flow that can be sent, since there can be n edges out of s, and each of them has at most U capacity. Each augmenting path sends at least 1 unit of flow, so this results in O(nU) augmenting paths. Finding an augmenting path can be done in O(m) time, which leads to the bound of O(nmU).It is somewhat like linear search whose time complexity increases as the number of elements in the array increases so as the max flow is going to be big it is going to be big problem for this algorithm. So Edmand Karp was better algorithm than Ford Fulkerson in which we are using BFS search to find the augmenting path. It is not guaranteed to terminate on real valued constraints. In 1972, J. Edmonds and R. M. Karp.observed that if the augmenting path found in the algorithm by Ford and Fulkerson alway is a shortest augmenting path, the maximum number of augmenting paths is O(nm). This algorithm runs in O(nm^2) time.

About the same time, in 1970, E. A. Dinic published another improvement over the algorithm by Ford and Fulkerson. The paper by Dinic also includes the algorithm by Edmonds and Karp, but Dinic includes additional techniques to reduce the running

time to O(n^2m). His idea was to remove some edges in the graph, to get a layer graph which contain all paths from s to t that have length k, where k is the length of the shortest augmenting path in G. He then finds all augmenting paths in this layer graph, which is called the blocking flow. After that, he calculates the residual network of the original graph augmented with the blocking flow. With this new residual network, he finds a new layer graph where k' > k. To find all paths in the layer graph, he used a depth first search. Many subsequent algorithms are based on this idea of using layer graphs, but have an optimized algorithm for nding the blocking flow.

So Dinic's algorithm is considered to be more optimised.For worst case when there are maximum possible edges[1] the plot for their running time complexity function.



USE OF ALGORITHM IN DIFFERENT WAYS:

1. Maximum Matchings in Bipartite Graphs
2. Vertex Capacities and Vertex-Disjoint Paths
3. Assignment Problems
4. Baseball Elimination
5. Project Selection

1

You can read more on its application :

https://courses.engr.illinois.edu/cs498dl1/sp2015/notes/24-maxflowapps.pdf

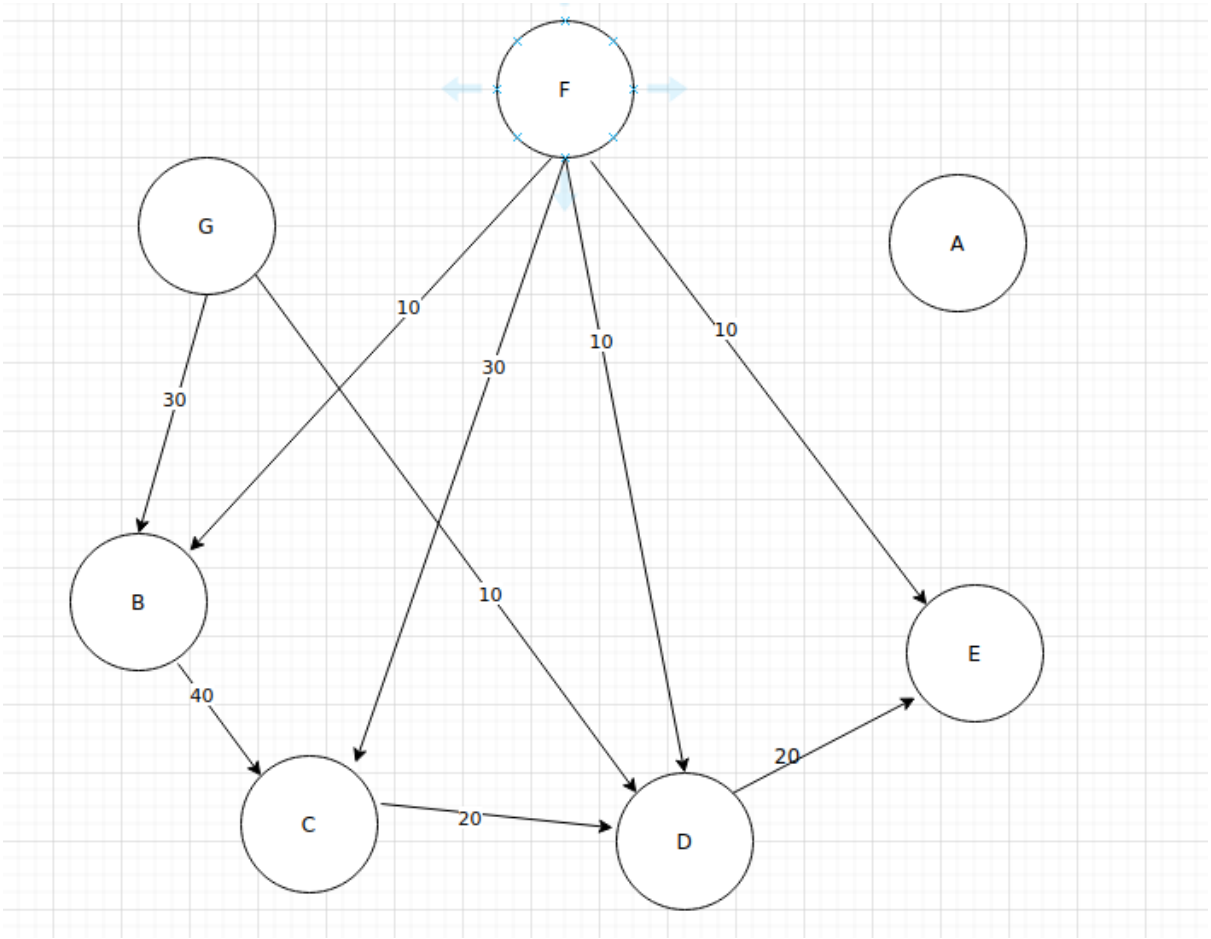
APPLICATION PART

APPLICATION OF MAX FLOW (DEBT SIMPLIFIER)

Consider a group of seven people namely A, B, C, D, E, F and G. They went out for a tour together and at the end of the tour realized that they have the following debts;

1. F owes $10 to B.
2. C owes $20 to D.
3. D owes $50 to E.
4. F owes $30 to C.
5. F owes $10 to D.
6. F owes $10 to E.
7. G owes $30 to B.
8. G owes $10 to D.
9. B owes $40 to C.

For better understanding, let's represent the above information in the form of a directed graph, which is as shown below.



Now, since the objective of the ‘Simplify Debts’ feature is to minimize the total number of payments made within the group, let’s try to try some algorithm for it.

- ➔ The first step is to determine the Net Change in Cash of each person in the group, which can be determined using the following formula,

Net Change in Cash = (Sum of Cash Inflow - Sum of Cash Outflow)

For example, C has a Net Change in Cash of $50, which is calculated as shown below.

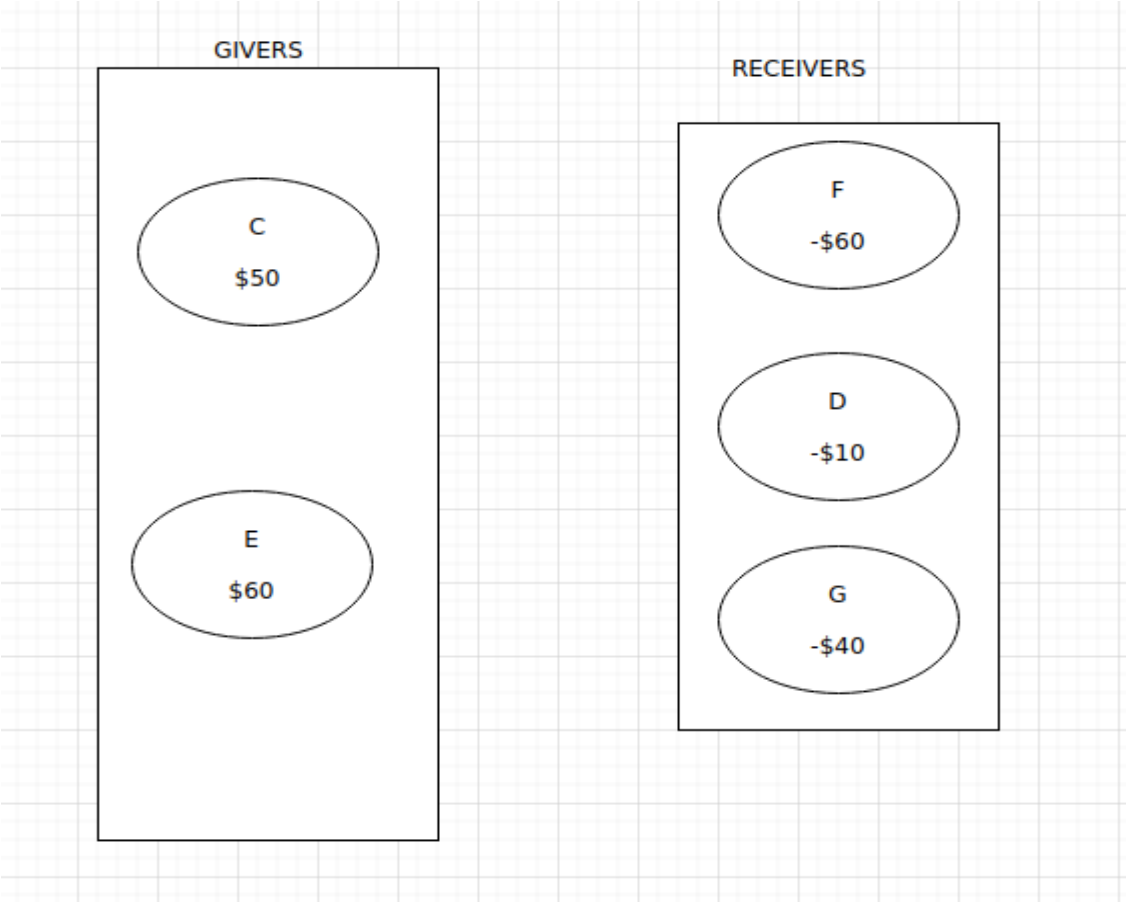
Net Change in Cash(C) = (Sum of Cash Inflow(C) — Sum of Cash Outflow(C))

- ➔ $((40+30) — (20)) = $50.

Thus, the overall Net Change in Cash is $0 for A, $0 for B, $50 for C, -$10 for D, $60 for E, -$60 for F and -$40 for G.

- ➔ Now we categorize them into two types of people, namely *Givers* (those who have extra cash, which is indicated by a positive value of *Net Change in Cash*) and *Receivers* (those who need extra cash, which is indicated by a negative value of *Net Change in Cash*). In the above example, C and E are

Givers, while D, F and G are *Receivers*. A and B have their debts already sorted out and hence they are neither *Givers* or *Receivers*.



Why is this problem **NP-Complete**?

In order to minimize the total payments to be made for resolving debts, *Givers* must transfer money only to *Receivers* and *Receivers* must receive money only from *Givers.* Also, it can be noted that the total money owed by *Givers* is always equal to the total money to be received by *Receivers*.

Since our objective is to minimize total payments to be made, we must ensure that each *Giver* transfers money to the least possible number of *Receivers*(Since otherwise he/she will end up making more transactions, which in turn will increase the total number of transactions made). This means that for any given debts, we have to always check for the scenario of each *Giver* making only one transaction(to some *Receiver*) in order to settle his/her debts(because then the total number of transactions will be equal to the number of *Givers* and hence will be the optimal solution). So, in the optimal scenario, each *Giver* will transfer money to exactly one *Receiver*. This, in turn, means that every *Receiver* should either receive the entire money from a *Giver* or not accept any amount from them altogether.

For example, let G1, G2, G3 and G4 be the amount owed by four *Givers* respectively. Also, let R1 and R2 represent the amount to be received by two *Receivers*. Now, for any given *Receiver*, he/she can either accept the entire amount G1 from the first Giver or not accept any amount altogether. Similarly, he/she can either accept the entire amount G2 from the second Giver or not accept any amount, and so on. This, in turn, means we are looking for a Subset of Givers that exactly adds up to a given Receiver's amount and we need to do this for all Receivers. This is nothing but the Sum of Subsets Problem, except that here we may be provided with more than one sum (depending upon the number of Receivers).

Now, what we were discussing until now is for the optimal case. It might also be the case that the best possible solution itself requires some subset of *Givers* to make more than one transaction. In that case, we need to do more than what we were planning to do for the optimal case, i.e. to check all possible ways of splitting the amount from *Givers*. This indicates that the debt simplification problem is at least as hard as the *Sum of Subsets Problem* and hence it is NP-Complete. Also, it means there is not only no polynomial-time solution to this problem but also that it will require an exponential number of steps to minimize the total number of payments.

Here, first and third rules are something that by default must be obeyed. But what is more interesting to us is the *Second Rule*, which says *'No one owes a person that they didn't owe before'.*

So the problem boils down to varying the amount being transferred on existing transactions without introducing newer ones**.** This algorithmically translates to the following,

Given a Directed Graph representing Debts (as shown in above Figure), change (if needed) the weights on the existing edges without introducing newer ones.

Now the question is how do we do it. This actually can be solved if we divide the problem into two halves as shown below,

1. *Will an existing edge(i.e. transaction) be part of the graph after simplifying debts?*
2. If it's present in the *graph after* simplifying *debts, then what will be the weight (i.e. amount) of it?*

The answer to the second question is to ***maximize the debt(i.e. weight) on the edge, so as to get rid of debts flowing along other paths between the same pair of vertices***. For example, let's have a look at *Figure 1* again. Here, if F transfers $20 to E, then F will not have to pay D anything and D will only have to pay $40 to E, thereby reducing the total transactions to be made from 3 to 2.

Now, in order to address the first question, i.e. to know if an edge will be part of the graph after simplifying debts, we will try all the edges one by one.

How do we maximize weight on an existing edge?

Once we select an edge, we can maximize its weight(i.e. debt) by using the Maximum-Flow Algorithm, which helps one determine maximum flow between a *source* and a *sink* in a given directed graph.

So, the algorithm to solve the problem is as mentioned below,

1. Feed the debts in the form of a directed graph (let's represent it by *G*) to the algorithm.
2. Select one of the non-visited edges, say *(u, v)* from the directed graph *G*.
3. Now with *u* as *source* and *v* as *sink*, run a maxflow algorithm to determine the maximum flow of money possible from *u* to *v*.
4. Also, compute the *Residual Graph* (let's represent it by *G'*), which indicates the additional possible flow of debts in the input graph after removing the flow of debts between *u* and *v*.
5. If maximum flow between *u* and *v (*let's represent it by *max-flow*) is greater than zero, then add an edge *(u, v)* with weight as *max-flow* to the *Residual Graph*.
6. Now go back to Step 1 and feed it the *Residual Graph G'*.
7. Once all the edges are visited, the *Residual Graph G'* obtained in the final iteration will be the one that has the minimum number of edges(i.e. transactions).

IMPLEMENTATION CODE LOGIC

The simplifier part of this is the implementation of the optimised case which is discussed above in which we have made two bipartite sets which are givers and receivers .

1. Firstly we take the number of people included in the share(expenses).

2. After that we calculate the net amount for every individual.

3. Now we have to look that who has credited maximum and debited maximum and find the minimum of the max

credited and debited amount now debit that amount to the max debit and credit that amount to the max creditor.

4. If the minimum of max credited and debited is equal to max credited amount then remove maximum creditor from the set of persons and recur for the remaining (n-1) persons.

5. If the minimum of max credited and debited is equal to max debited amount then remove maximum creditor from the set of persons and recur for the remaining (n-1) persons.

BASIC CP GUIDE FOR GRAPHS

Graph Representation

There are several ways to represent graphs in algorithms. The choice of a data structure depends on the size of the graph and the way the algorithm processes it. We will discuss three representation below:

1. Adjacency List

   In the adjacency list representation, each node x in the graph is assigned an adjacency list that consists of nodes to which there is an edge from x . Adjacency lists are the most popular way to represent graphs, and most algorithms can be efficiently implemented using them. Some convenient ways for this can be:

```
std::vector<int> adj_list[MN];
```

   The constant MN is chosen so that all adjacency list can be stored. Every element of the adj_list array points to a vector which can be of any size. For undirected graph we push vertices for both the inputs like:

```
int u, v;
cin >> u >> v;
adj_list[u].push_back(v);
adj_list[v].push_back(u);
```

   For problems in which we have undirected weighted graph, this can look like:

```
int n, m;
cin >> n >> m;
vector<pii> adj[n];

int a, b, wt;
for (int i = 0; i < m; i++)
{
    cin >> a >> b >> wt;
    adj[a].push_back(make_pair(b, wt));
    adj[b].push_back(make_pair(a, wt));
}
```

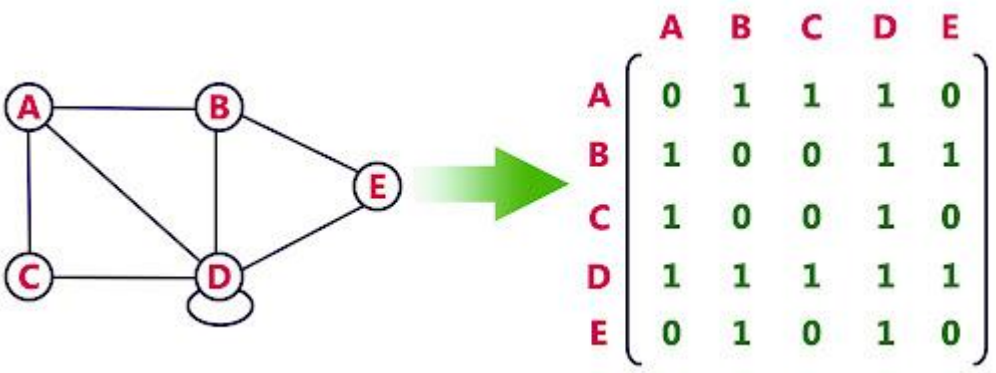
We can similarly do for directed graph but instead of pushing in both sides we will push only in one. The benefit of using adjacency lists is that we can efficiently find the nodes to which we can move from a given node through an edge. For example, the following loop goes through all nodes to which we can move from node s :

```
for (int i = 0; i <= N; i++)
{
    for (auto s : adj_list[i]){
        cout << s << " ";
    }
}
```

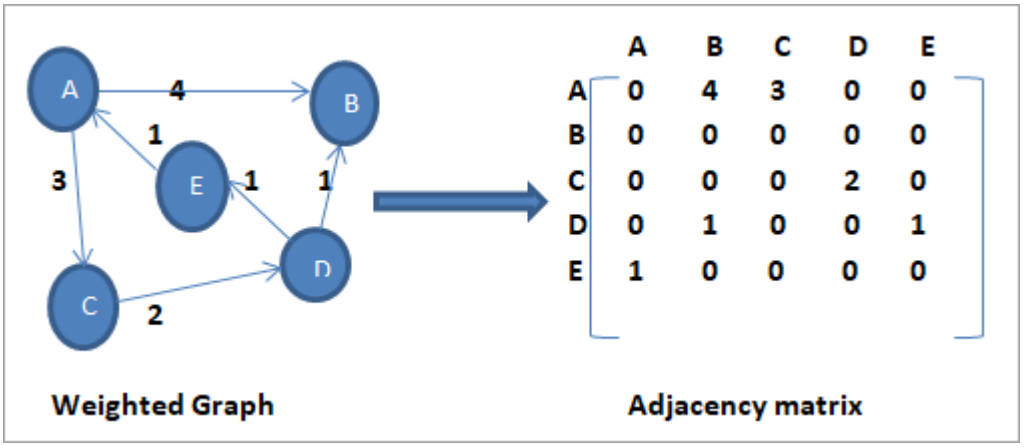
2. Adjacency Matrix

An adjacency matrix is a two-dimensional array that indicates which edges the graph contains. We can efficiently check from an adjacent matrix if there is an edge between two nodes. The matrix can be stored as an array where each value adj_mat[x][y] indicates whether the graph contains an edge from node x to node y. If the edge is Included in the graph, then adj_mat[x][y] = 1, and otherwise Adj_mat[x][y] = 0.

```
int adj_mat[N+1][N+1];
```



For weighted graph, adj_mat[x][y] = weight otherwise adj_mat[x][y] =0. For example,



The drawback of the adjacency matrix representation is that the matrix contains n^2 elements, and usually most of them a zero. That's why we Use an adjacency list most of the time.

3. Edge List

An edge list contains all edges of a graph in some order. This is a convenient way to represent a graph if the algorithm process all edges of the graph and it is not needed to find edges that start at a given node.

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

vector<tuple<int, int, int>> EDGES;

struct Edge
{
   int a, b, w;

};
```

First representation is for unweighted graph and other two are for

Weighted graph. Use of this representation is done when you have to Specifically deal with edges, like deleting some edges. Sometimes we have to make a vector for storing degree of the vertices and when we add any edge then we increase the degree of both the vertices and same When an edge is deleted . For example:

```
vector<int> degree(N+1);
while (M--)
{

    int u, v;
    cin >> u >> v;
    adj_list[u].push_back(v);
    adj_list[v].push_back(u);
    degree[u]++;
    degree[v]++;
}
```

DFS

Depth-first search (DFS) is a straightforward graph traversal technique. The algorithm begins at a starting node, and proceeds to all other nodes that are reachable from the starting node using the edges of the graph.
We will discuss three implementations DFS in graphs, DFS in trees and DFS in grids. All three are very important regarding CP.

1. DFS in Graphs
   The function assumes that the graph is stored as adjacency lists in an array and also maintains an array visited that keeps track of the visited nodes. Initially, each array value is false , and when the search arrives at node s , the value of visited [s] becomes true .

```
void Dfs(int node)
{
    visited[node]=1;
    for (int child :a[node])
    {
       if(!visited[child])
       dfs(child);
    }

}
```

```
//bipartite graph
int col[1000];
bool bipartite(int v,int color)
{
    //node and its child should have differen color so if same
    //then it is not bipartite
    visited[v]=true;
    col[v]=color;
    for(int child:adj_list[v])
    {
        if(visited[child]==false)
        {
            if(bipartite(child,color^1)==false)
              return false;

        }
        else
        if(col[v]==col[child])
        return false;
    }
    return true;

}
```

With the help of DFS we can find the graph is bipartite or node, by providing colors (binary) to the nodes and checking its child nodes if their color is the same or not. It is used when we have to divide nodes in two sets to deal with some problem.

Cycle Detection can also be done using DFS .

```cpp
//cycle detection dfs   //if kisi ka child phle se visited h to vo cycle milega vha pr
//dusra method by finding edges and nodes if node ==n and edhes != n-1 to cycle h
bool cycle(int node,int par)
{
    visited[node]=true;
    for(int child:adj_list[node])
    {
        if(visited[child]==false)
        {
            if(cycle(child,node)==true)
            return true;
        }
        else{
            if(child!=par)
            return true;
        }

    }
       return false;
}
```

Most of the basic Graph problems use DFS , the thing that matters most of the time is how to implement it. You should have good practise to have an idea after reading the problem. You will be using different data structures like maps, sets, multisets to implement them.

2. DFS in Trees

Some properties/definitions of trees:

- A graph is a tree iff it is connected and contains NNN nodes and N−1N-1N−1 edges
- A graph is a tree iff every pair of nodes has exactly one simple path between them
- A graph is a tree iff it is connected and does not contain any cycles

DFS will be applied similarly , but there are many things we can do in trees using this. To find the diameter of the tree,single source shortest path,longest path in tree, subtree size etc.

We can see their implementations below:

Longest path and diameter of the tree are the same. In the below implementation we will first find the farthest node from the start node or root node, when we get the farthest node , we make that node to be the parent node and calculate the node which is farthest from modified root node using again the dfs function.

```cpp
int N, dis, e;
vector<int> adj[mxN];

void dfs(int u, int p, int depth) {
    for (int v : adj[u]) {
        if (v == p) continue;
        dfs(v, u, depth + 1);
    }

    if (depth > dis) {
        dis = depth;
        e = u;
    }
}
```

Subtree Size problem , This will calculate the size of subtree of every node. It means from the level of the node , how many nodes are there whose parent is the node(parent) .

```cpp
vector<int> children[SZ];
int subtree_size[SZ], depth[SZ];

void dfs_size(int node) {
    subtree_size[node] = 1; // This one represents the root of `node's` subtree
    // (which would be `node` itself)
    for (int child : children[node]) {
        depth[child] = depth[node]+1; // not needed for this problem
        dfs_size(child);
        subtree_size[node] += subtree_size[child];
        // Add `node's` children's subtrees to the size of `node's` subtree
    }
}
```

3. DFS on grids

Flood fill is an algorithm that identifies and labels the connected component that a particular cell belongs to in a multidimensional array. We define two directions arrays where we can move from one cell . For example , for a grid where one can move to left,right,up and down positions . The arrays be like

```cpp
int dx[] = {1,-1,0,0};
int dy[] = {0,0,1,-1};
```

For grid , in which we can go to any neighbour , the arrays are like

```
int dx[] = {1,-1,0,0,1,-1,-1,1};
int dy[] = {0,0,1,-1,1,-1,1,-1};
```

We make a separate function to check if the cell is valid or not, it will be invalid if it is out of bound or some specific condition according to the question. For example,

```
bool IsValid(int xx, int yy)
{
return (xx >= 0 and xx < n and yy >= 0 and yy < m and grid[xx][yy] == '.') ? true : false;
}
```

The DFS function for the grid is simple , we start from the start cell, and then check every neighbour array directions for a given condition.

PROBLEMS ON DFS

1. https://codeforces.com/problemset/problem/377/A
2. https://codeforces.com/problemset/problem/1598/A
3. https://codeforces.com/problemset/problem/1332/C
4. https://codeforces.com/problemset/problem/1249/B2
5. https://codeforces.com/problemset/problem/510/B
6. https://codeforces.com/problemset/problem/687/A

BFS

Breadth-first search (BFS) visits the nodes in increasing order of their distance from the starting node. Thus, we can calculate the distance from the starting node to all other nodes using breadth-first search.Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

We use Queue data structure for its implementation which is the First In First Out working algorithm. We pop the parent node and push its all child nodes in the queue.

In an unweighted graph , bfs is going to help in finding the shortest path nodes from the root node. As we traverse level wise , so all the nodes at one level are at the same

distance from the root node. We will assume the first distance of the root node to be zero and keep updating other distances accordingly.

```cpp
vector<int> dist(10000,-1);

void bfs(int node){
    queue<int> q;
    q.push(node);
    visited[node]=1;
    dist[node]=0;
    while(!q.empty()){
        int curr=q.front();
        q.pop();
        for(int child:a[curr]){
            if(!visited[child]){
                q.push(child);
                visited[child]=1;
                dist[child]=dist[curr]+1;
            }
        }
    }
}
```

In this dist vector stores the distance of all the nodes of the graph from the root node. Problems like:

Finding connected components in an undirected graph , least number of moves to reach some state , previous problem in grid like chess , or some square game, shortest cycle in directed unweighted graph.

0/1 BFS

A 0/1 BFS finds the shortest path in a graph where the weights on the edge can only be 0 or 1 and runs in O(V+E) usine a dequeue. Dequeue is a data structure present in the standard library of Cpp which allows to push and pop from both sides.

Basically this type of problem is given as grid where we can move in four sides from one cell so the distance will be 0 or 1 for any cell from the root cell.

```
int dx[]={0,0,1,-1};
int dy[]={1,-1,0,0};
int Visited[100][100];
int n, m;
int Dist[100][100];
void BFS(int x, int y)
{
    queue<pii> q;
    q.push({x,y});
    Visited[x][y]=1;
    Dist[x][y]=0;
    while(!q.empty()){
        pii curr=q.front();
        q.pop();
        for(int i=0;i<4;i++){
            int nx=curr.first+dx[i];
            int ny=curr.second+dy[i];
            if(nx>=0 && nx<n && ny>=0 && ny<n && !Visited[nx][ny]){
                q.push({nx,ny});
                Visited[nx][ny]=1;
                Dist[nx][ny]= Dist[curr.first][curr.second]+1;
            }
        }
    }
}
```

PROBLEMS ON BFS

1. https://cses.fi/problemset/task/1193
2. http://www.usaco.org/index.php?page=viewproblem2&cpid=620
3. https://codeforces.com/contest/59/problem/E
4. https://csacademy.com/contest/archive/task/bfs-dfs
5. https://cses.fi/problemset/task/1670/
6. http://www.usaco.org/index.php?page=viewproblem2&cpid=575
7. https://oj.uz/problem/view/IOI09_mecho

CONNECTED COMPONENTS

We can find connected components using both DFS and BFS .

DFS: The first round will start from the first node and all the nodes in the first connected component will be traversed (found). Then we find the first unvisited node of the remaining nodes, and run Depth First Search on it, thus finding a second connected component. And so on, until all the nodes are visited.

```cpp
vector<int> comp;
void Dfs(int v)
{
    visited[v] = true;
    comp.push_back(v);
    for(auto to : a[v])
    {
        if(!visited[to])
            Dfs(to);
    }
}

void COMPS_dfs()
{
    for (int i = 0; i < n; ++i)
        visited[i] = false;

    for (int i = 0; i < n; ++i)
        if (!visited[i])
        {
            comp.clear();
            Dfs(i);
            cout << "Component:";
            for (size_t j = 0; j < comp.size(); ++j)
                cout << ' ' << comp[j];
            cout << endl;
        }
}
```

BFS: Similar implementation for BFS as it is done for DFS.

DSU:

DSU is a very important data structure which tells its meaning by its name; it joins the disjoint sets which are linked (given input) .*The Disjoint Set Union (DSU) data structure allows you to add edges to an initially empty graph and test whether two vertices of the graph are connected.*

```
int parent[mex];
int size[mex];
void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}

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

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}
```

Using set data structure for every element's parent , then set will store only unique elements , and the size of the set data structure (set.size()) will be the number of components.

```
void num_comp()
{
    set<int> comp;
    for(int i=1;i<= mex;i++){
        comp.insert(parent[i]);
    }
    cout << "Number of components: ";
    cout << comp.size() << endl;
}
```

These implementations are for undirected graphs , there is a separate algorithm for directed graphs or We call Strongly Connected Components . The algorithm is Tarjan algorithm that basically involves two times DFS.

PROBLEMS ON CONNECTED COMP./DSU

1. https://cses.fi/problemset/task/1666
2. https://cses.fi/problemset/task/1676
3. http://www.usaco.org/index.php?page=viewproblem2&cpid=669
4. https://csacademy.com/contest/archive/task/mountain-time
5. http://www.usaco.org/index.php?page=viewproblem2&cpid=1042
6. https://codeforces.com/problemset/problem/771/A
7. https://codeforces.com/problemset/problem/902/B

TOPOLOGICAL SORT

An ordering of vertices in a directed acyclic graph that ensures that a node is visited before every node it has a directed edge to.

A topological sort of a directed acyclic graph is a linear ordering of its vertices such that for every directed edge u→v from vertex u to vertex v, u comes before v in the ordering.

There are two common ways to topologically sort, one involving DFS and the other involving BFS.

DFS

When started from start node v, it tries to run along all edges outgoing from v . It fails to run along the edges for which the opposite ends have been visited previously, and runs along the rest of the edges and starts from their ends.

Thus by the time the call dfs(v) is ended , all vertices that are reachable from v either directly (adjacent) or indirectly are already visited by the search. Therefore, if at the time of exit from dfs(v) we add vertex v to the beginning of a certain list , in the end this list will store a topological ordering of all vertices.

```cpp
int N;                                  // Number of nodes
vector<int> graph[100000], top_sort; // Assume that this graph is a DAG
bool visited[100000];

void dfs(int node)
{
    for (int i : graph[node])
    {
        if (!visited[i])
        {
            visited[i] = true;
            dfs(i);
        }
    }
    top_sort.push_back(node);
}

void compute()
{
    for (int i = 0; i < N; i++)
    {
        if (!visited[i])
        {
            visited[i] = true;
            dfs(i);
        }
    }
    reverse(begin(top_sort), end(top_sort));
    // The vector `top_sort` is now topologically sorted
}
```

BFS

KAHN'S ALGO: According to the algo , first we insert all the start nodes who have no incoming edges in a vector and one by one apply BFS for that vector of nodes. If in this way we make such nodes whose incoming node is 0 , then we push it in the vector.

```cpp
vector<int> res;

vector<int> arr[1000];

void kahnsalgo(int n)
{
    queue<int> q;
    for (int i = 1; i <= n; i++)
    {
        /* code */
        if (in[i] == 0)
            q.push(i);
    }
    while (!q.empty())
    {
        /* code */
        int curr = q.front();
        res.push_back(curr);

        q.pop();
        for (int node : arr[curr])
        {
            in[node]--;
            if (in[node] == 0)
                q.push(node);
        }
    }
}
```

Problems like

1. Operation System deadlock detection
2. Dependency resolution
3. Sentence Ordering
4. Critical Path Analysis
5. Course Schedule problem
6. Other applications like manufacturing workflows, data serialization and context-free grammar.
7. Finding cycle in a graph

PROBLEMS ON TOPOLOGICAL SORT

1. https://cses.fi/problemset/task/1681
2. https://open.kattis.com/problems/quantumsuperposition
3. https://codeforces.com/contest/919/problem/D
4. https://codeforces.com/problemset/problem/510/C
5. https://cses.fi/problemset/task/1757

SHORTEST PATH

DIJKSTRA

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. This algorithm can not be used to find shortest distance when there are negative edges. For that , we have a separate algorithm.

For better optimisation of the algorithm , we use priority queue data structure for its implementation. Dijkstra's algorithm performs n iterations, On each iteration it selects an unmarked vertex v with the lowest value d[v], marks it and checks all the edges (v,to) attempting to improve the value d[to].

The function takes the starting vertex s and two vectors that will be used as return values. We use a vector of pairs in which the first value is the distance and second is the node. We first insert the start node in the queue and then start iterating and pushing according to the priority . The node with less distance will be at the front as the priority queue sorts the nodes according to the distance.

Problems

1. https://codeforces.com/problemset/problem/20/C
2. https://codeforces.com/problemset/problem/59/E
3. https://www.spoj.com/problems/CCHESS/

```cpp
//dijkstra

 ll inf = 1e18;
void dijkstra(vector<vector<pii>> g, int s, vector<ll> &d)
{
   typedef pair<ll, int> T;
   priority_queue<T, vector<T>, greater<T>> q;
   for (q.push({0, s}); !q.empty(); q.pop())
   {
      ll curd = q.top().first;
      int i = q.top().second;
      if (d[i] != inf)
         continue;
      d[i] = curd;
      trav(p, g[i]) q.push({curd + p.second, p.first});
   }
}
```

BELLMAN FORD

Unlike the Dijkstra algorithm, this algorithm can also be applied to graphs containing negative weight edges . However, if the graph contains a negative cycle, then, clearly, the shortest path to some vertices may not exist (due to the fact that the weight of the shortest path must be equal to minus infinity); however, this algorithm can be modified to signal the presence of a cycle of negative weight, or even deduce this cycle.

The algorithm consists of several phases. Each phase scans through all edges of the graph, and the algorithm tries to produce **relaxation** along each edge (a,b) having weight c. Relaxation along the edges is an attempt to improve the value d[b] using value d[a]+c. In fact, it means that we are trying to improve the answer for this vertex using edge (a,b) and current response for vertex a.
It is claimed that n-1 phases of the algorithm are sufficient to correctly calculate the lengths of all shortest paths in the graph .
With some more modification in the implementation , we can the shortest path instead of the length of the shortest path.
For that, let's create another array p[0…n−1], where for each vertex we store its "predecessor", i.e. the penultimate vertex in the shortest path leading to it. In fact, the shortest path to any vertex a is a shortest path to some vertex p[a], to which we added a at the end of the path.

The below implementation is for negative edges too .

```
struct Ed
{
    int a, b, w;

  int s()
    { return a < b ? a : -a; }
};
struct Node
{
    ll dist = inf;
    int prev = -1;
};
```

```
void bellmanFord(vector<Node> &nodes, vector<Ed> &eds, int s)
{
    nodes[s].dist = 0;
    sort(all(eds), [](Ed a, Ed b)
        { return a.s() < b.s(); });

    int lim = sz(nodes) / 2 + 2; // /3+100 with shuffled vertices
    rep(i, 0, lim) trav(ed, eds)
    {
        Node cur = nodes[ed.a], &dest = nodes[ed.b];
        if (abs(cur.dist) == inf)
            continue;
        ll d = cur.dist + ed.w;
        if (d < dest.dist)
        {
            dest.prev = ed.a;
            dest.dist = (i < lim - 1 ? d : -inf);
        }
    }
    rep(i, 0, lim) trav(e, eds)
    {
        if (nodes[e.a].dist == -inf)
            nodes[e.b].dist = -inf;
    }
}
```

It is easy to see that the Bellman-Ford algorithm can endlessly do the relaxation among all vertices of this cycle and the vertices reachable from it. Therefore, if you do not limit the number of phases to n−1, the algorithm will run indefinitely, constantly improving the distance from these vertices. Hence we obtain the criterion for presence of a cycle of negative weights reachable for source vertex v: after (n−1)th phase, if we run algorithm for one more phase, and it performs at least one more relaxation, then the graph contains a negative weight cycle that is reachable from v; otherwise, such a cycle does not exist.

So we can detect if there is a negative cycle present in the graph through this algorithm.

Problems:

1. https://cses.fi/problemset/task/1197
2. https://www.eolymp.com/en/problems/1453

FLOYD WARSHELL

This is a three loop algorithm , so its complexity is more than other two algorithms but it is considered to be a good one when we have to find the shortest distance of every vertex from another vertex . It makes a 2D array for storing the distances. The key idea of the algorithm is to partition the process of finding the shortest path between any two vertices to several incremental phases.

```
const ll inf = 1e18;

void floydWarshall(vector<vector<ll>> &m)
{
   int n = sz(m);
   rep(i, 0, n) m[i][i] = min(m[i][i], 0LL);
   rep(k, 0, n) rep(i, 0, n) rep(j, 0, n) if (m[i][k] != inf && m[k][j] != inf)
   {
      auto newDist = max(m[i][k] + m[k][j], -inf);
      m[i][j] = min(m[i][j], newDist);
   }

   rep(k, 0, n)
    if (m[k][k] < 0)
     rep(i, 0, n)
     rep(j, 0, n)
     if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
}
```

Problems:

1. https://codeforces.com/problemset/problem/295/B
2. https://www.spoj.com/problems/CHICAGO/
3. https://codeforces.com/problemset/problem/21/D
4. https://www.facebook.com/codingcompetitions/hacker-cup/2021/qualification-round/problems/A2

MST Problems:

1. http://www.usaco.org/index.php?page=viewproblem2&cpid=531
2. https://codingcompetitions.withgoogle.com/kickstart/round/0000000000436140/000000000068c2c3
3. https://codeforces.com/problemset/problem/1513/D
4. https://www.hackerrank.com/contests/w31/challenges/spanning-tree-fraction/problem
5. https://codeforces.com/contest/888/problem/G
