Introduction to Algorithms
Science Honors Program (SHP)
Session 5

Christian Lim

Saturday, March 23, 2024

Slide deck in github

- You may get to the link by:
  - https://github.com/yongwhan/
  - => yongwhan.github.io
  - => columbia
  - => shp
  - => session 5 slide

Overview

- **Dynamic Programming (con't)**
- **Shortest Paths**
- Break #1 (5-minute)
- **Minimum Spanning Trees**
- **Lowest Common Ancestor**
- Break #2 (5-minute)
- **Flows**

CodeForces Columbia SHP Algorithms Group

- While I take the attendance, please join the following group: https://codeforces.com/group/lfDmo9iEr5

- We will be using them in the last portion of the session today!

Attendance

- Let's take a quick attendance before we begin!

Tree DP

- Topological Sort (Warm-Up)
- Longest Path in (Weighted) Directed Acyclic Graph

Topological Sorting

- In a directed acyclic graph (DAG),

Topological Sorting

- In a directed acyclic graph (DAG), put into queue all nodes with indegree zero.

Topological Sorting

- In a directed acyclic graph (DAG), put into queue all nodes with indegree zero.

- Each time a node is dequeued, decrement their children's indegree by 1 and anytime it hits 0, put in that node into queue.

Topological Sorting

- In a directed acyclic graph (DAG), put into queue all nodes with indegree zero.
- Each time a node is dequeued, decrement their children's indegree by 1 and anytime it hits 0, put in that node into queue.
- Rinse and repeat!

Topological Sorting

- Of course, you can do it using DFS too!

Topological Sorting: Implementation

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

Topological Sorting: Implementation

```
void topological_sort() {
  visited.assign(n, false);
  ans.clear();
  for (int i = 0; i < n; ++i)
    if (!visited[i])
      dfs(i);
  reverse(ans.begin(), ans.end());
}
```

Hamiltonian Flight (CSES 1690)

- There are n cities and m flight connections between them. You want to travel from Syrjälä to Lehmälä so that you visit each city exactly once. How many possible routes are there?

Discuss for few minutes!

Solution Idea?

- **Bitmask DP**

- Let's cover the exact implementation details later!

DP is a large topic!

- There are other topics like **DP optimization** (Divide and Conquer, Knuth, and Convex Hull Trick, etc). We may cover them later, if we have time!

Shortest Paths

- **Shortest Paths**
  - BFS/DFS
  - Dijkstra
  - Floyd-Warshall
  - Bellman-Ford

Shortest Path

- In a graph, you'd like to find a shortest path (e.g., a path with a minimum total cost) from a source node `s` to a target node `t`.

Shortest Path

- In a graph, you'd like to find a shortest path (e.g., a path with a minimum total cost) from a source node `s` to a target node `t`.

- Multiple settings are possible:

Shortest Path

- In a graph, you'd like to find a shortest path (e.g., a path with a minimum total cost) from a source node **`s`** to a target node **`t`**.

- Multiple settings are possible:
  - Unweighted:

Shortest Path

- In a graph, you'd like to find a shortest path (e.g., a path with a minimum total cost) from a source node `s` to a target node `t`.

- Multiple settings are possible:
  - Unweighted: **BFS/DFS**

Shortest Path

- In a graph, you'd like to find a shortest path (e.g., a path with a minimum total cost) from a source node `s` to a target node `t`.

- Multiple settings are possible:
    - Unweighted: BFS/DFS
    - Non-negative weights:

Shortest Path

- In a graph, you'd like to find a shortest path (e.g., a path with a minimum total cost) from a source node `s` to a target node `t`.

- Multiple settings are possible:
  - Unweighted: BFS/DFS
  - Non-negative weights: **Dijkstra**

Shortest Path

- In a graph, you'd like to find a shortest path (e.g., a path with a minimum total cost) from a source node `s` to a target node `t`.

- Multiple settings are possible:
  - Unweighted: BFS/DFS
  - Non-negative weights: Dijkstra
  - Negative weights:

Shortest Path

- In a graph, you'd like to find a shortest path (e.g., a path with a minimum total cost) from a source node **`s`** to a target node **`t`**.

- Multiple settings are possible:
    - Unweighted: BFS/DFS
    - Non-negative weights: Dijkstra
    - Negative weights: **Bellman-Ford**

Shortest Path

- In a graph, you'd like to find a shortest path (e.g., a path with a minimum total cost) from a source node `s` to a target node `t`.

- Multiple settings are possible:
    - Unweighted: BFS/DFS
    - Non-negative weights: Dijkstra
    - Negative weights: Bellman-Ford
    - All Pairs:

Shortest Path

- In a graph, you'd like to find a shortest path (e.g., a path with a minimum total cost) from a source node `s` to a target node `t`.

- Multiple settings are possible:
    - Unweighted: BFS/DFS
    - Non-negative weights: Dijkstra
    - Negative weights: Bellman-Ford
    - All Pairs: **Floyd-Warshall**

BFS

- Queue!

BFS: Implementation

```
vector<vector<int>> adj;  // adjacency list
int n; // number of nodes
int s; // source vertex

queue<int> q;
vector<bool> used(n);
vector<int> d(n), p(n);
```

BFS: Implementation

```
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;
    }
  }
}
```

DFS

- Stack!

DFS: Implementation

```
vector<vector<int>> adj; // adjacency list
int n; // number of vertices

vector<bool> visited;

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

Dijkstra

- Priority Queue

Dijkstra: Implementation

- Priority Queue

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;
```

Dijkstra: Implementation

```
void dijkstra(int s,
              vector<int> & d,
              vector<int> & p) {
  int n = adj.size();
  d.assign(n, INF); p.assign(n, -1); d[s] = 0;
  set<pair<int, int>> q; q.insert({0, s});
```

Dijkstra: Implementation

```
  while (!q.empty()) {
    int v = q.begin()->second;
    q.erase(q.begin());
    for (auto edge : adj[v]) {
      int to = edge.first, len = edge.second;
      if (d[v] + len < d[to]) {
        q.erase({d[to], to}); d[to] = d[v] + len;
        p[to] = v; q.insert({d[to], to});
      }
    }
  }
}
```

Bellman-Ford

- Relaxation!

Bellman-Ford: Implementation

```
vector<int> d(n, INF);
d[v] = 0;
for (int i = 0; i < n - 1; i++) {
  bool any = false;
  for (Edge e : edges)
    if (d[e.a] < INF)
      if (d[e.b] > d[e.a] + e.cost) {
        d[e.b] = d[e.a] + e.cost; any = true;
      }
  if (!any) break;
}
```

Bellman-Ford: Implementation

```
vector<int> d(n, INF);
d[v] = 0;
for (int i = 0; i < n - 1; i++) {
  bool any = false;
  for (Edge e : edges)
    if (d[e.a] < INF)
      if (d[e.b] > d[e.a] + e.cost) {
        d[e.b] = d[e.a] + e.cost; any = true;
      }
  if (!any) break;
}
```

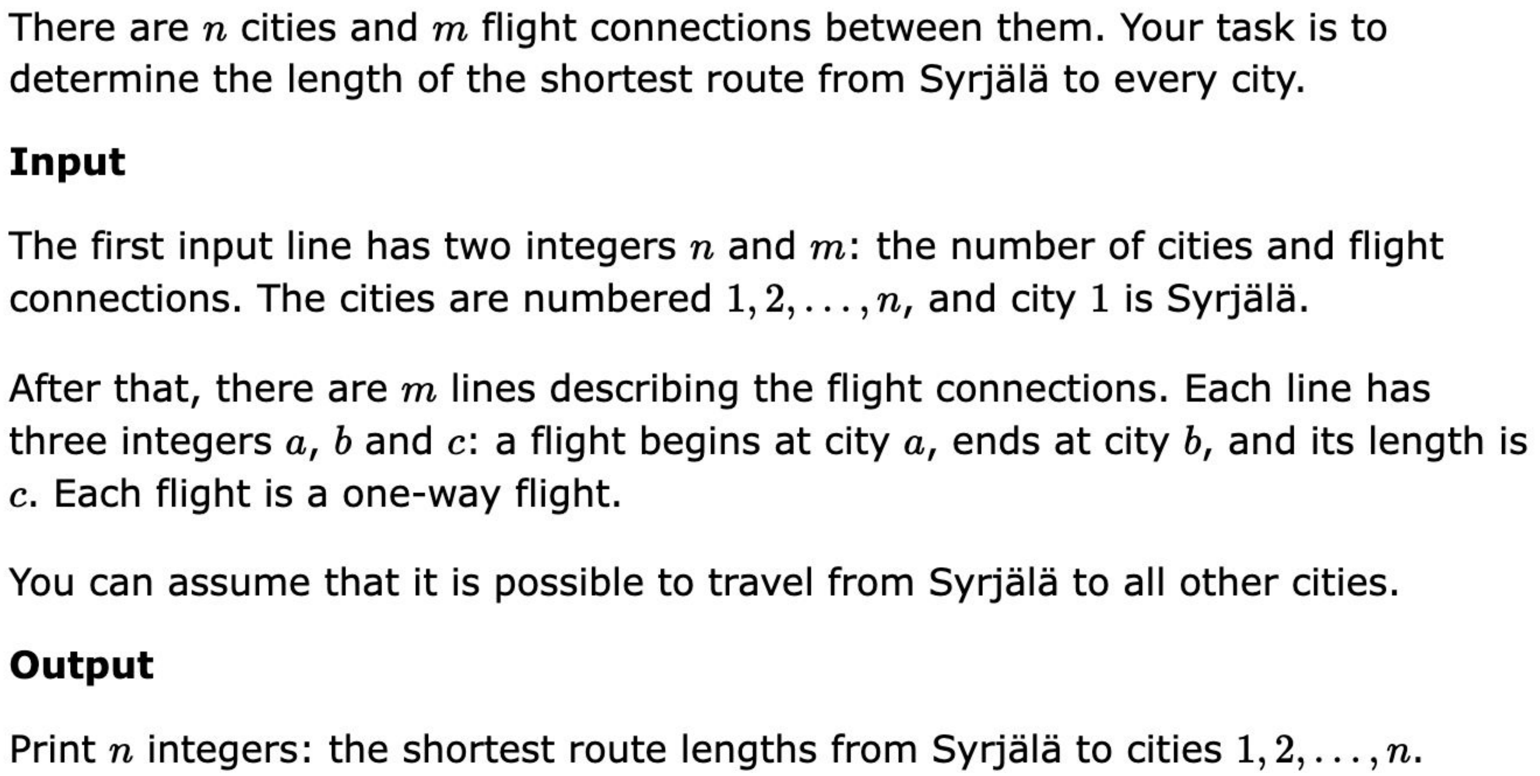
Floyd-Warshall

- Triple for loops!

Floyd-Warshall: Implementation

```
for (int k = 0; k < n; k++)
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
```

Shortest Routes I (CSES 1671)



There are n cities and m flight connections between them. Your task is to determine the length of the shortest route from Syrjälä to every city.

Input

The first input line has two integers n and m: the number of cities and flight connections. The cities are numbered $1, 2, \dots, n$, and city 1 is Syrjälä.

After that, there are m lines describing the flight connections. Each line has three integers a, b and c: a flight begins at city a, ends at city b, and its length is c. Each flight is a one-way flight.

You can assume that it is possible to travel from Syrjälä to all other cities.

Output

Print n integers: the shortest route lengths from Syrjälä to cities $1, 2, \dots, n$.

Discuss for few minutes!

Solution Idea?

- **Dijkstra!**

Practice Problems

- **Dynamic Programming** (available in CodeForces group!)
  - https://codeforces.com/problemset/problem/1566/C
  - https://codeforces.com/problemset/problem/919/B
  - https://codeforces.com/problemset/problem/522/A
  - https://codeforces.com/problemset/problem/1037/C
  - https://codeforces.com/problemset/problem/1108/D
  - https://codeforces.com/problemset/problem/1389/C
  - https://codeforces.com/problemset/problem/1288/C
  - https://codeforces.com/problemset/problem/1091/D
  - https://codeforces.com/problemset/problem/645/D

Practice Problems

- **Shortest Path** (available in CodeForces group!)
  - https://codeforces.com/contest/59/problem/E
  - https://codeforces.com/contest/796/problem/D
  - https://codeforces.com/contest/821/problem/D
  - https://codeforces.com/contest/758/problem/E
  - https://codeforces.com/contest/761/problem/E
  - https://codeforces.com/contest/767/problem/C
  - https://codeforces.com/problemset/problem/20/C
  - https://codeforces.com/problemset/problem/59/E
  - https://codeforces.com/contest/25/problem/C

Reference

- **For more details, please take a look at the following tutorials:**
  - https://usaco.guide/gold/dp-bitmasks?lang=cpp;
  - https://usaco.guide/gold/dp-trees?lang=cpp;
  - https://cp-algorithms.com/graph/breadth-first-search.html
  - https://cp-algorithms.com/graph/depth-first-search.html
  - https://cp-algorithms.com/graph/dijkstra.html
  - https://cp-algorithms.com/graph/bellman_ford.html
  - https://cp-algorithms.com/graph/all-pair-shortest-path-floyd-warshall.html

BREAK #1

Minimum Spanning Trees

- Kruskal's
- Prim's

Minimum Spanning Trees

- **Prim**
  -
- **Kruskal**
  -

Minimum Spanning Trees

- **Prim**
  - After choosing any initial node, add the edge with the least cost!
- **Kruskal**
  -

Minimum Spanning Trees

- **Prim**
  - After choosing any initial node, add the edge with the least cost!
- **Kruskal**
  - Sort edge weights and unite nodes if they are not already connected.

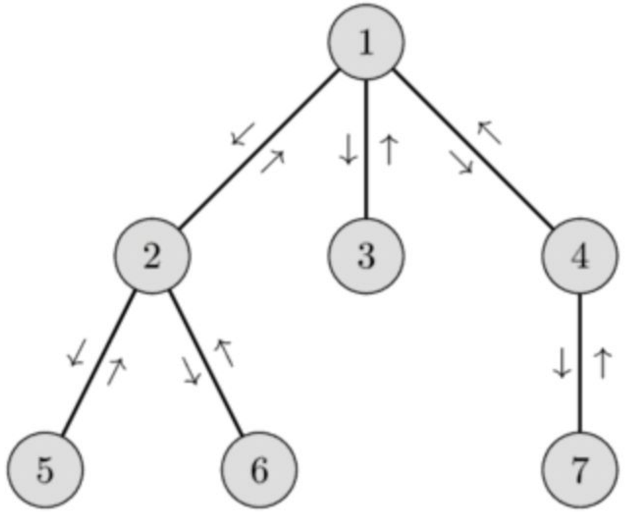
Kruskal's: Implementation

```
int n;
vector<Edge> edges;
int cost = 0;
vector<Edge> result;
parent.resize(n);
rank.resize(n);
for (int i = 0; i < n; i++)
  make_set(i);
```

Kruskal's: Implementation

```
sort(edges.begin(), edges.end());
for (Edge e : edges) {
  if (find_set(e.u) != find_set(e.v)) {
    cost += e.weight;
    result.push_back(e);
    union_sets(e.u, e.v);
  }
}
```

Lowest Common Ancestor (LCA): Euler Tour first!



Vertices:	1	2	5	2	6	2	1	3	1	4	7	4	1
Heights:	1	2	3	2	3	2	1	2	1	2	3	2	1

Lowest Common Ancestor (LCA): Example

- The tour starting at vertex 6 and ending at 4 we visit the vertices [6, 2, 1, 3, 1, 4].
- Among those vertices the vertex 1 has the **lowest height**.
- Therefore, LCA(6, 4) = 1!

Lowest Common Ancestor (LCA): Example

- In general, once we pre-process the nodes using Euler tour, you can do a **range minimum query**.

- We know this can be solved using **segment tree** (or, **sparse table**, if the tree is fixed!)

Lowest Common Ancestor (LCA): Implementation

```
struct LCA {
  vector<int> height, euler, first, segtree;
  vector<bool> visited;
  int n;
```

Lowest Common Ancestor (LCA): Implementation

```
LCA(vector<vector<int>> &adj, int root = 0) {
  n = adj.size();
  height.resize(n);
  first.resize(n);
  euler.reserve(n * 2);
  visited.assign(n, false);
  dfs(adj, root);
  int m = euler.size();
  segtree.resize(m * 4);
  build(1, 0, m - 1);
}
```

Lowest Common Ancestor (LCA): Implementation

```
void dfs(vector<vector<int>> &adj, int node, int h=0) {
  visited[node] = true;
  height[node] = h;
  first[node] = euler.size();
  euler.push_back(node);
  for (auto to : adj[node])
    if (!visited[to]) {
      dfs(adj, to, h + 1);
      euler.push_back(node);
    }
}
```

Lowest Common Ancestor (LCA): Implementation

```
void build(int node, int b, int e) {
  if (b == e) {
    segtree[node] = euler[b];
  } else {
    int mid = (b + e) / 2;
    build(node << 1, b, mid);
    build(node << 1 | 1, mid + 1, e);
    int l = segtree[node<<1], r = segtree[node<<1|1];
    segtree[node] = (height[l] < height[r]) ? l : r;
  }
}
```

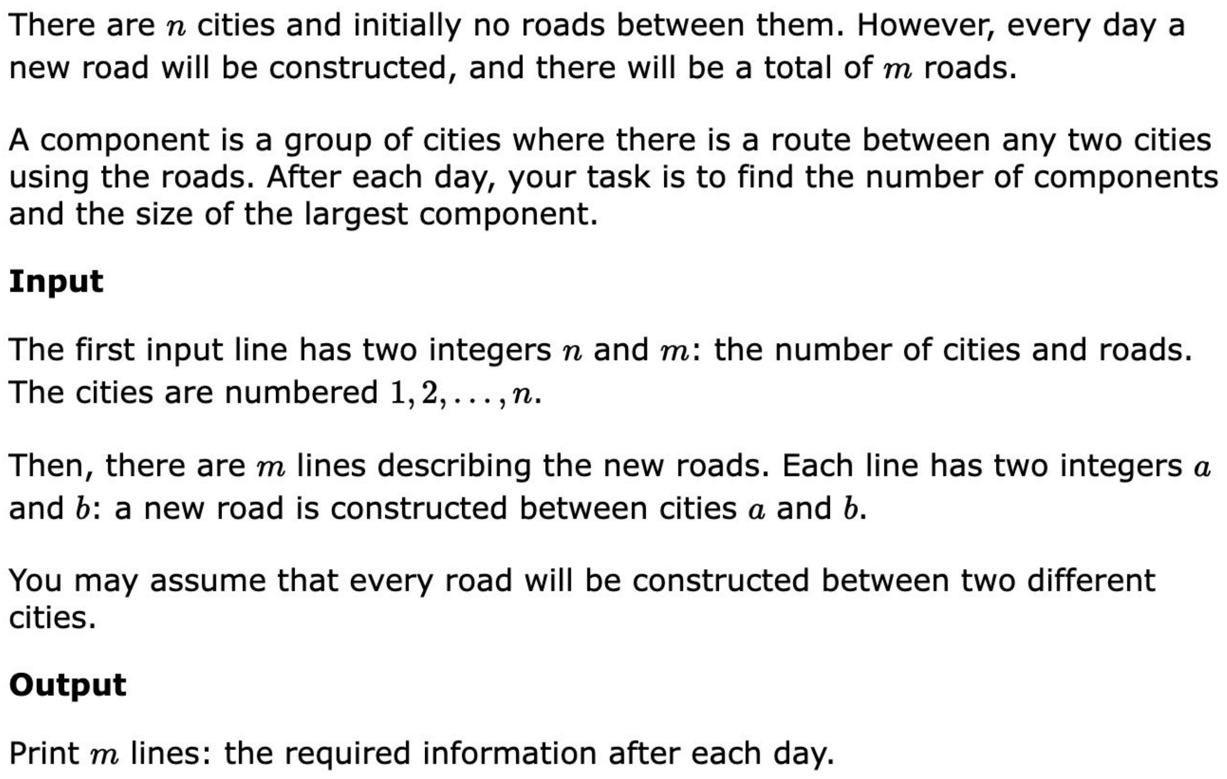
Lowest Common Ancestor (LCA): Implementation

```
int query(int node, int b, int e, int L, int R) {
  if (b > R || e < L) return -1;
  if (b >= L && e <= R) return segtree[node];
  int mid = (b + e) >> 1;
  int left = query(node << 1, b, mid, L, R);
  int right = query(node << 1 | 1, mid + 1, e, L, R);
  if (left == -1) return right;
  if (right == -1) return left;
  return height[left] < height[right] ? left : right;
}
```

Lowest Common Ancestor (LCA): Implementation

```
  int lca(int u, int v) {
    int left = first[u], right = first[v];
    if (left > right)
      swap(left, right);
    return query(1, 0, euler.size() - 1, left, right);
  }
};
```

Road Construction (CSES 1676)



There are n cities and initially no roads between them. However, every day a new road will be constructed, and there will be a total of m roads.

A component is a group of cities where there is a route between any two cities using the roads. After each day, your task is to find the number of components and the size of the largest component.

Input

The first input line has two integers n and m: the number of cities and roads. The cities are numbered $1, 2, \dots, n$.

Then, there are m lines describing the new roads. Each line has two integers a and b: a new road is constructed between cities a and b.

You may assume that every road will be constructed between two different cities.

Output

Print m lines: the required information after each day.

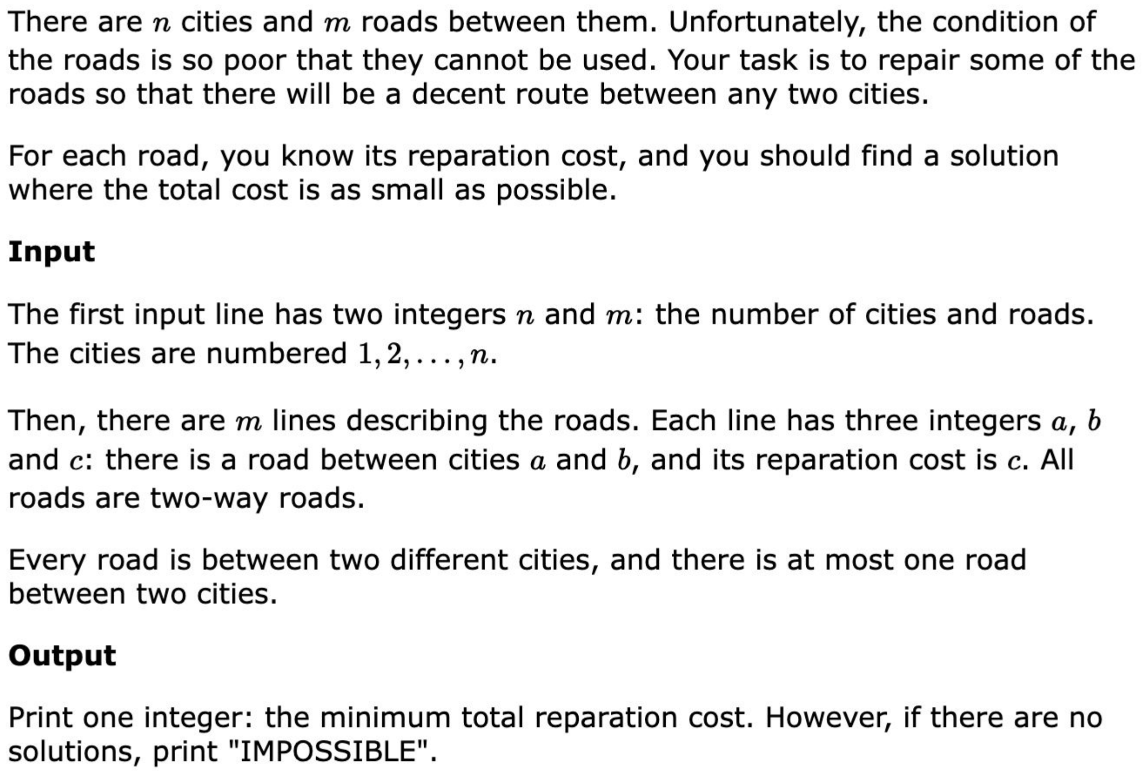
Discuss for few minutes!

Solution Idea?

- Disjoint Set Union!

- https://usaco.guide/problems/cses-1676-road-construction/solution

Road Reparation (CSES 1675)



There are n cities and m roads between them. Unfortunately, the condition of the roads is so poor that they cannot be used. Your task is to repair some of the roads so that there will be a decent route between any two cities.

For each road, you know its reparation cost, and you should find a solution where the total cost is as small as possible.

Input

The first input line has two integers n and m: the number of cities and roads. The cities are numbered $1, 2, \dots, n$.

Then, there are m lines describing the roads. Each line has three integers a, b and c: there is a road between cities a and b, and its reparation cost is c. All roads are two-way roads.

Every road is between two different cities, and there is at most one road between two cities.

Output

Print one integer: the minimum total reparation cost. However, if there are no solutions, print "IMPOSSIBLE".

Discuss for few minutes!

Solution Idea?

- **Kruskal!**

Practice Problems

- **Minimum Spanning Trees** (available in CodeForces group!)
  - https://codeforces.com/contest/160/problem/D
  - https://codeforces.com/problemset/problem/32/C
  - https://codeforces.com/problemset/problem/598/D
  - https://codeforces.com/problemset/problem/744/A
  - https://codeforces.com/contest/17/problem/B

Practice Problems

- **Lowest Common Ancestor** (available in CodeForces group!)
  - https://codeforces.com/problemset/problem/472/D
  - https://codeforces.com/contest/733/problem/F
  - https://codeforces.com/contest/828/problem/F
  - https://codeforces.com/contest/832/problem/D
  - https://codeforces.com/contest/855/problem/D

Reference

- **For more details, please take a look at the following tutorials:**
  - https://cp-algorithms.com/graph/mst_kruskal_with_dsu.html
  - https://cp-algorithms.com/graph/mst_prim.html

BREAK #2

Network Flow: Real-life Example

- We are given a directed graph, where each vertex represents a city and each directed edge represents a one-way road from one city to another.
- Suppose we want to send trucks from city s to city t and the capacity of each directed edge represents the number of trucks that can go on that road every hour.

Photo Credit

Network Flow: Real-life Example

- We are given a directed graph, where each vertex represents a city and each directed edge represents a one-way road from one city to another.
- Suppose we want to send trucks from city s to city t and the capacity of each directed edge represents the number of trucks that can go on that road every hour.

- **What is the maximum number of trucks that we can send from s to t every hour?**

Flow Network

- A **network** is a directed graph G with vertices V and edges E combined with a function c, which assigns each edge e a non-negative integer value, the **capacity** of e.
- Such a network is called a **flow network**, if we additionally label two vertices, one as **source** and the other as **sink**.

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 conditions:
  - The flow of an edge cannot exceed the capacity: $f(e) \leq c(e)$.
  - 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): $\Sigma_v f((v, u)) = \Sigma_w f((u, w))$.
- The source vertex s only has outgoing flows.
- The sink vertex t only has incoming flows.
- $\Sigma_v f((v, t)) = \Sigma_w f((s, w))$.

Flow Network

- The value of the flow of a network is the sum of all the flows that get produced in the source s, or equivalently to the sum of all the flows that are consumed by the sink t.
- Formally, val(f), the **value** of a flow f, is defined as:

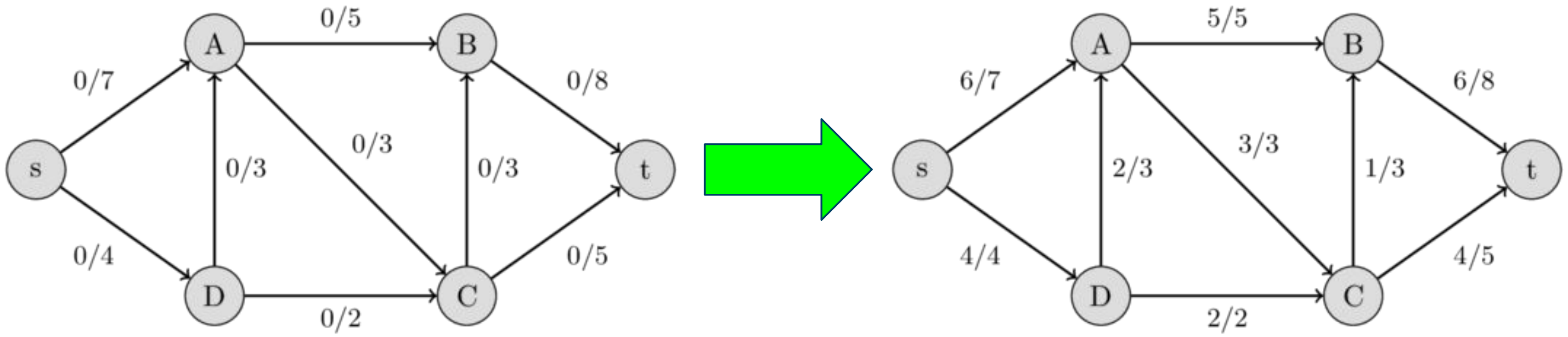
$$\Sigma_{\text{e out of s}} f(e) - \Sigma_{\text{e in to s}} f(e).$$

- A **maximum flow** is a flow with the maximum possible value of val(f).

- **Problem: Find this maximum flow of a flow network!**

Ford-Fulkerson Method: Example

- The first value of each edge represents the flow, which is initially 0, and the second value represents the capacity.



Ford-Fulkerson Method (1956)

- A **residual capacity** of an directed edge is the capacity minus the flow.
  - For each edge (u, v), we can create a reverse edge (v, u) with capacity 0 such that f((v, u)) = -f((u, v)).
  - This also defines the residual capacity for all the reversed edges.
- We can create a **residual network** from all these edges, which is just a network with the same vertices and edges, but we use the residual capacities as capacities.

Ford-Fulkerson Method (con't)

- We set the flow of each edge to 0.
- We look for an **augmenting path** from s to t.
  - An augmenting path is a simple path in the residual graph by always taking the edges whose residual capacity is **positive**.
- If such a path is found then we can increase the flow along these edges.
  - Let C be the smallest residual capacity of the edges in the path. Then we increase the flow by updating f((u, v)) += C and f((v, u)) -= C for every edge (u, v) in the path.
- Else, terminate! (A flow found is maximum).

Ford-Fulkerson Method: Augmenting Path?

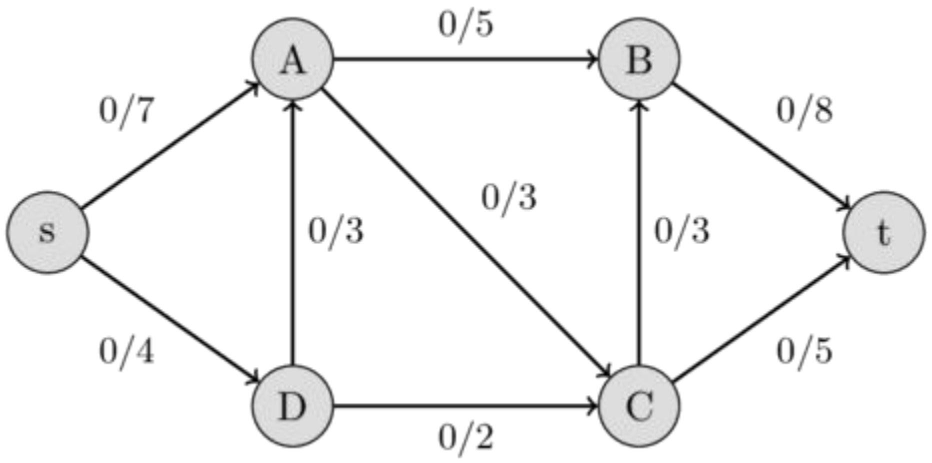
- Ford-Fulkerson method **does not specify** how to find an augmenting path. Possible approaches are using *Depth-First Search (DFS)* or *Breadth-First Search (BFS)*, both of which work in **O(E)**.

Ford-Fulkerson Method: Augmenting Path? (con't)

- **Integral** capacities
  - For each augmenting path, the flow of the network increases by at least 1. So, the complexity of Ford-Fulkerson is O(EF) where F is the maximum flow of the network (but, usually *much faster* in practice)!
- **Rational** capacities
  - The algorithm will terminate but the complexity is not bounded.
- **Irrational** capacities
  - The algorithm might never terminate and might not even converge to the maximum flow.

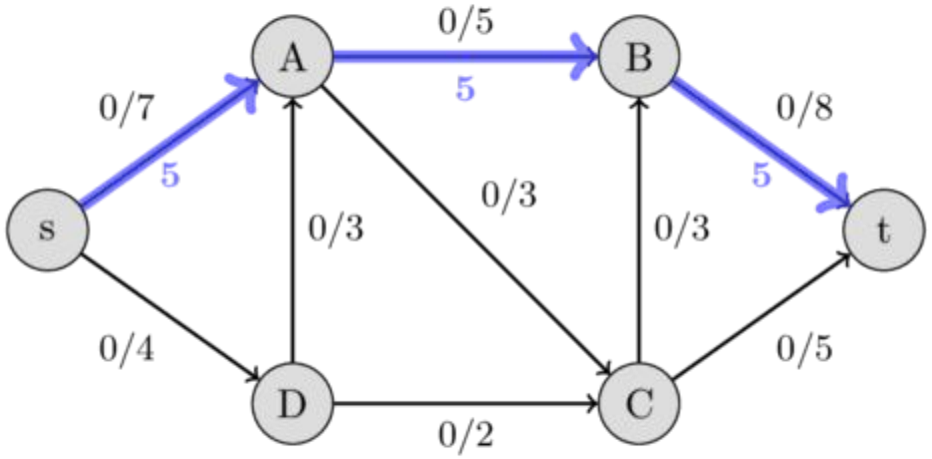
Ford-Fulkerson Method: Example

flow = 0



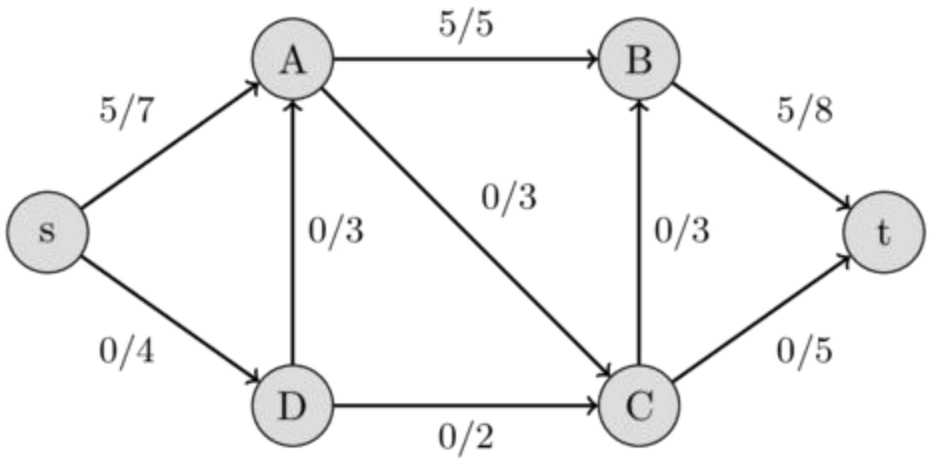
Ford-Fulkerson Method: Example

flow = 0



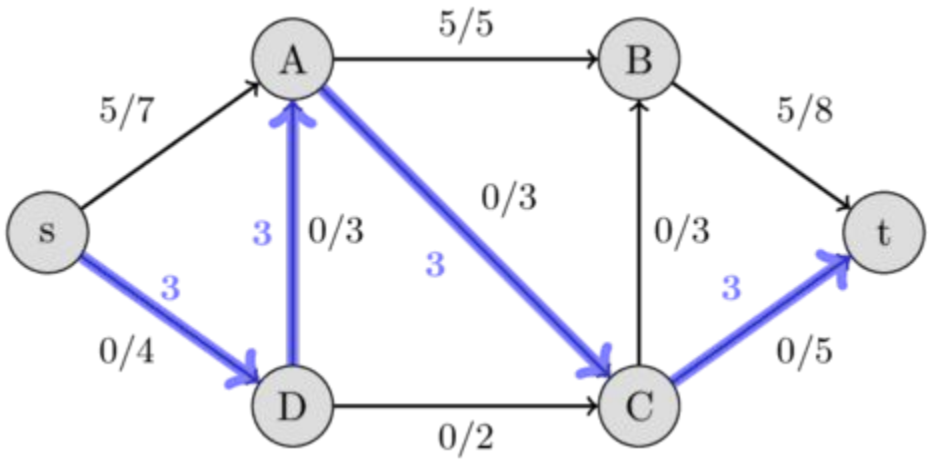
Ford-Fulkerson Method: Example

flow = 5



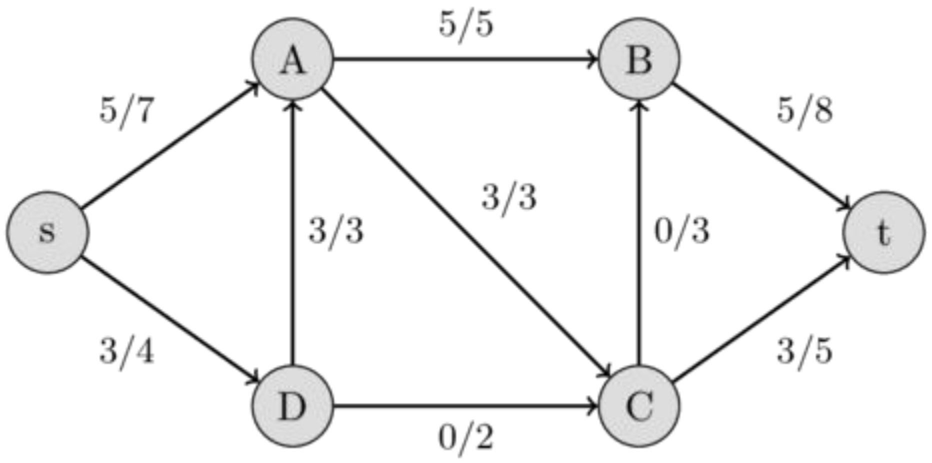
Ford-Fulkerson Method: Example

flow = 5



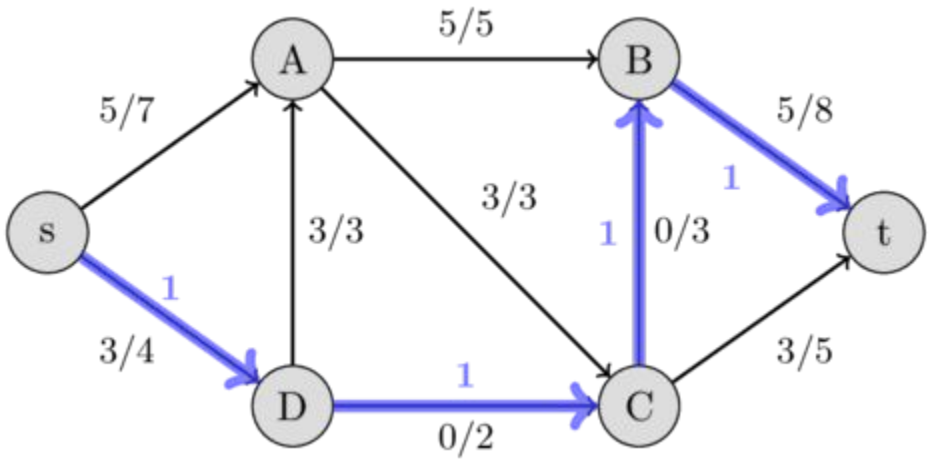
Ford-Fulkerson Method: Example

flow = 8



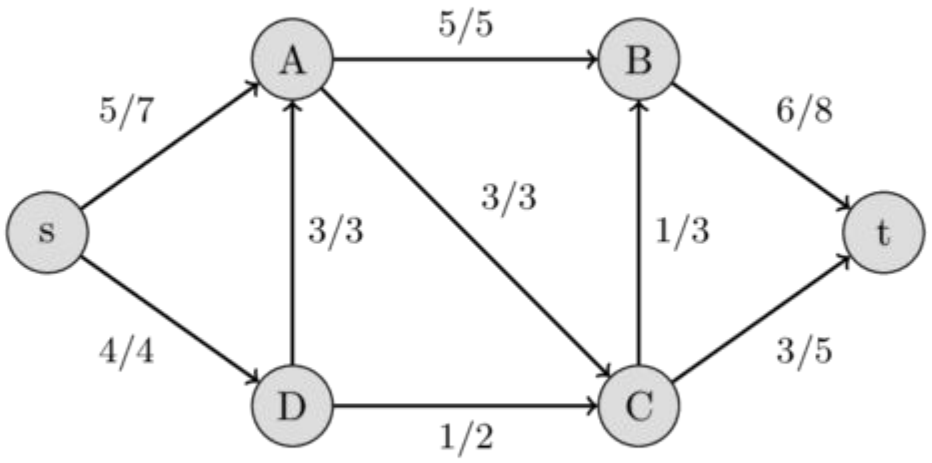
Ford-Fulkerson Method: Example

flow = 8



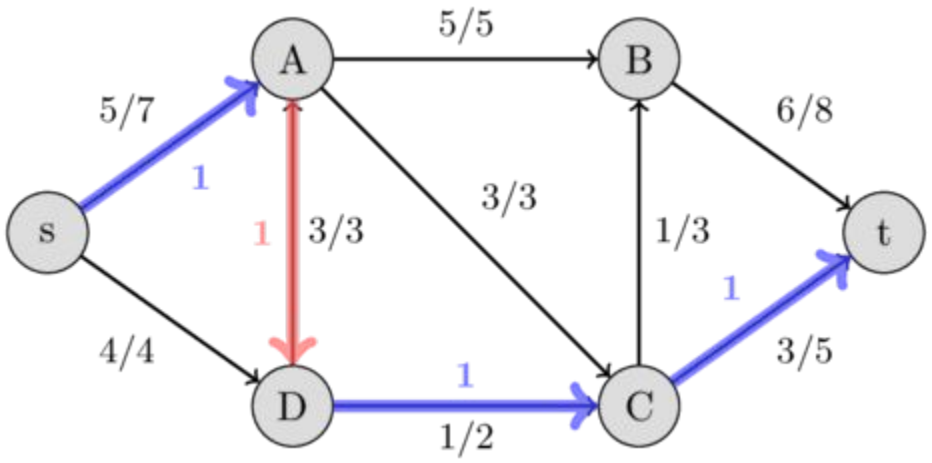
Ford-Fulkerson Method: Example

flow = 9



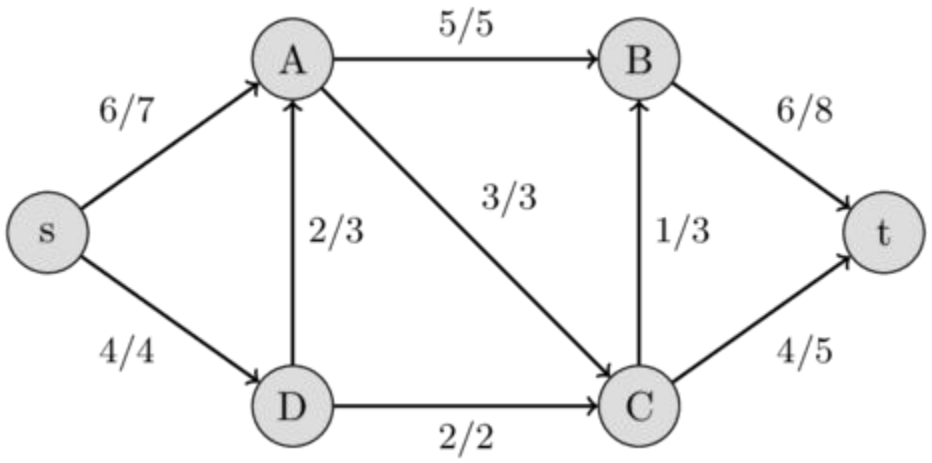
Ford-Fulkerson Method: Example

flow = 9



Ford-Fulkerson Method: Example

flow = **10**



Edmonds-Karp Algorithm (1972)

- **Edmonds-Karp algorithm** is just an implementation of the Ford-Fulkerson method that uses BFS for finding augmenting paths.

- First published by Yefim Dinitz in 1970; later, independently published by Jack Edmonds and Richard Karp in 1972.

- The complexity can be given *independently* of the maximum flow. The algorithm runs in **$O(VE^2)$** time (yes! even for irrational capacities).

Edmonds-Karp Algorithm (1972)

- **Intuitions**
  - Every time we find an augmenting path. one of the edges becomes saturated and the distance from the edge to s will be longer if it appears again in an augmenting path later.
  - The path length is bounded by V.

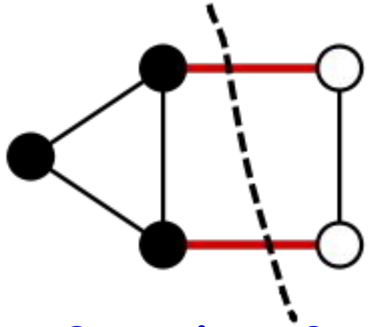
Integral Flow Theorem

- Suppose a given graph has each of its capacity integral. Then, there exists a maximum flow f where every flow value f(e) is an integer.

s-t cut and its capacity

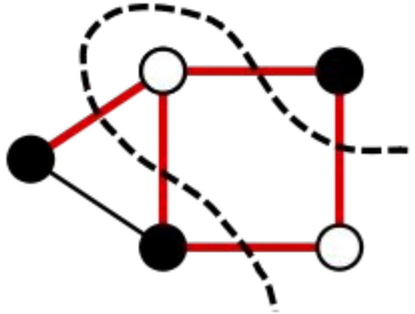
- An **s-t cut** is a partition (A, B) of the vertices with s ∊ A and t ∊ B.
- The capacity of a cut, **cap(A, B)**, is the sum of capacities of the edges from A to B.

Minimum Cut



Capacity = 2

Maximum Cut



Capacity = 5

Flow Value Lemma

- Let f be any flow and let (A, B) be any cut. Then,

$$val(f) = \Sigma_{e \text{ out of } A} f(e) - \Sigma_{e \text{ in to } A} f(e).$$

Flow Weak Duality Lemma

- Let f be any flow and let (A, B) be any cut. Then,

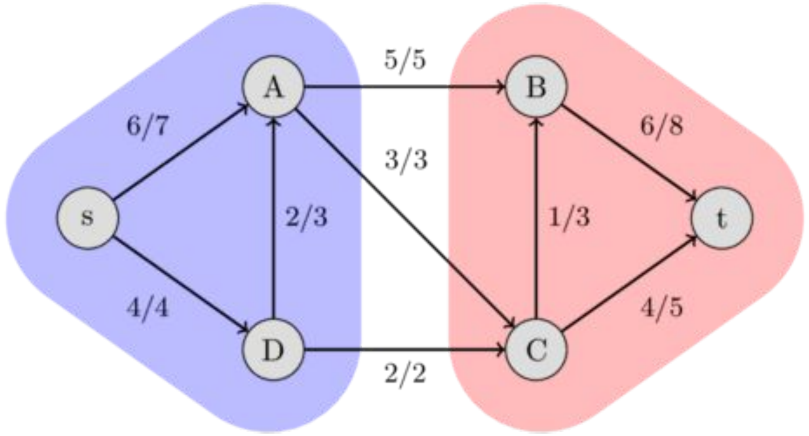
$$\mathrm{val}(f) \leq \mathrm{cap}(A, B).$$

Flow Certificate of Optimality

- Let f be a flow and let (A, B) be any cut. If val(f) = cap(A, B) then f is a maximum flow and (A, B) is a minimum cut.

Max-flow Min-cut Theorem

- The maximum value of an s-t flow is equal to the minimum capacity over all s-t cuts.



It is quite rare to actually use Ford Fulkerson in CP!

- But, without **Ford Fulkerson** and an intuition behind an **augmenting path**, it is quite difficult to understand **Dinic**, which is why we covered it first!

- Now, let's dive into **Dinic**!

Definitions

- A **residual network** G^R of network G is a network which contains two edges for each edge $(v, u) \in G$:
    - (v, u) with capacity $c^R_{vu} = c_{vu} - f_{vu}$
    - (u, v) with capacity $c^R_{uv} = f_{vu}$

Definitions

- 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.
  - Note that a blocking flow is not necessarily maximal.

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

Dinic's Algorithm

- The algorithm consists of several phases.
- On each phase:
  - Construct the layered network of the residual network of G.
  - Find an arbitrary blocking flow in the layered network and add it to the current flow.

Number of Phases

- The algorithm terminates in less than V phases.

Finding Blocking Flow

- In order to find the blocking flow on each iteration, we may simply try pushing flow with DFS (Depth-First Search) from s to t in the layered network while it can be pushed.
- In order to do it more efficiently, we must remove the edges which cannot be used to push anymore.
- We can keep a **pointer** in each vertex which points to the next edge which can be used.

Finding Blocking Flow (con't)

- A single DFS run takes O(k + V) time, where k is the number of pointer advances on this run.
- Over all runs, a number of pointer advances cannot exceed E.
- A total number of runs would not exceed E, as every run saturates at least one edge.
- So, a total running time of finding a blocking flow is **O(VE)**.

Dinic's Complexity

- Since there are less than V phases, the total time complexity is **$O(V^2E)$**.

Implementations in C++: *FlowEdge* struct

```
struct FlowEdge {
   int v, u;
   long long cap, flow = 0;
   FlowEdge(int v, int u, long long cap) :
     v(v), u(u), cap(cap) {}
};
```

Implementations in C++: *Dinic* struct

```
struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;
    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n); level.resize(n); ptr.resize(n);
    }
```

Implementations in C++: *Dinic* struct

```
    void add_edge(int v, int u, long long cap);
    bool bfs();
    long long dfs(int v, long long pushed);
    long long flow();
};
```

Implementations in C++: *flow* function

```
long long flow() {
  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;
}
```

Implementations in C++: *bfs* function

```cpp
bool bfs() {
  while (!q.empty()) {
    int v = q.front(); q.pop();
    for (int id : adj[v]) {
      if (edges[id].cap-edges[id].flow < 1) continue;
      if (level[edges[id].u] != -1) continue;
      level[edges[id].u] = level[v] + 1;
      q.push(edges[id].u);
    }
  }
  return level[t] != -1;
}
```

Implementations in C++: *dfs* function

```
long long dfs(int v, long long pushed) {
  if (pushed == 0) return 0;
  if (v == t) return pushed;
  for (int& cid = ptr[v]; cid < (int)adj[v].size();
       cid++) {
    int id = adj[v][cid], u = edges[id].u;
    if (level[v] + 1 != level[u] ||
        edges[id].cap - edges[id].flow < 1)
      continue;
```

Implementations in C++: *dfs* function

```
    long long tr = dfs(u,
      min(pushed, edges[id].cap - edges[id].flow));
    if (tr == 0) continue;
    edges[id].flow += tr;
    edges[id ^ 1].flow -= tr;
    return tr;
  }
  return 0;
}
```

Implementations in C++: *add_edge* function

```
void add_edge(int v, int u, long long cap) {
    edges.emplace_back(v, u, cap);
    edges.emplace_back(u, v, 0);
    adj[v].push_back(m);
    adj[u].push_back(m + 1);
    m += 2;
}
```

Here is a full implementation in one page

- https://cp-algorithms.com/graph/dinic.html#implementation

Practice Problems

- **Network Flow**
  - https://codeforces.com/contest/498/problem/c
  - https://codeforces.com/contest/1288/problem/f
  - https://cses.fi/problemset/task/1694
  - https://cses.fi/problemset/task/1695
  - https://cses.fi/problemset/task/1696
  - https://cses.fi/problemset/task/1711
  - https://open.kattis.com/problems/maxflow
  - https://open.kattis.com/problems/mincut
  - https://open.kattis.com/problems/tomography

References

- **cp-algorithms**
  - https://cp-algorithms.com/graph/edmonds_karp.html
  - https://cp-algorithms.com/graph/dinic.html

Again, CodeForces Columbia SHP Algorithms Group

- Please join the following group: **https://codeforces.com/group/lfDmo9iEr5**

No Class Next Week (March 30)! Ad Hoc+ on April 6!

- Due to Easter, there will be **no class** on March 30!
- On April 6, we will cover graph algorithms:
  - **Ad Hoc**
  - **Combinatorics**

Slide Deck

- You may **always** find the slide decks from:
  - https://github.com/yongwhan/yongwhan.github.io/blob/master/columbia/shp

THANK YOU