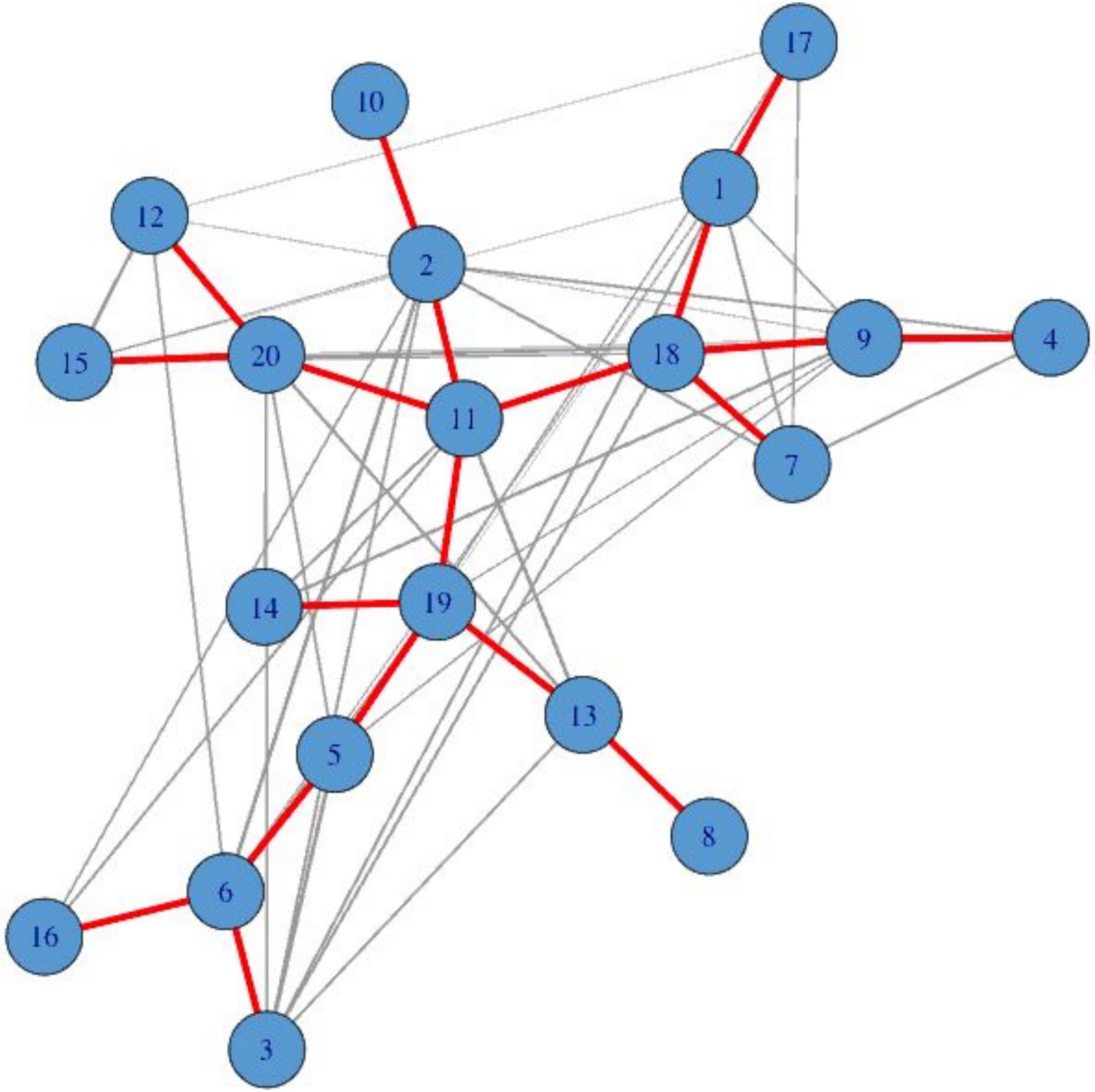
Graphs Overview for Competitive Programming 2

MST



Minimum Spanning Tree (MST)

- Input: Undirected weighted graph
- Output: A spanning tree with minimal weight

- Properties:
  - The heaviest edge in a cycle will not be in any MST (aka “Red rule”)
  - The lightest edge in a cut will be in every MST (aka “Blue rule”)
  - All MSTs have the same number of edges of each weight
    - If the weights are unique, there is a unique MST
  - Finding a maximum spanning tree is equivalent (just negate the weights)

Minimum Spanning Tree (MST)

- Prim and Kruskal suggested well-known algorithms for finding an MST

	Prim	Kruskal
Description	Grows a single component. At each step adding the lightest edge touching it.	Go over all edges by increasing weight. If adding the edge does not close a cycle, add it.
Implementation	Maintain a priority queue with the nodes adjacent to the component, along with the weight of the corresponding candidate edges. Take the minimum at each step.	Maintain a Union-Find data structure containing the components. An edge closes a cycle iff both the nodes belong to the same Union-Find component.
Time complexity		

Implementing Kruskal

$O(|E|\log|V|)$

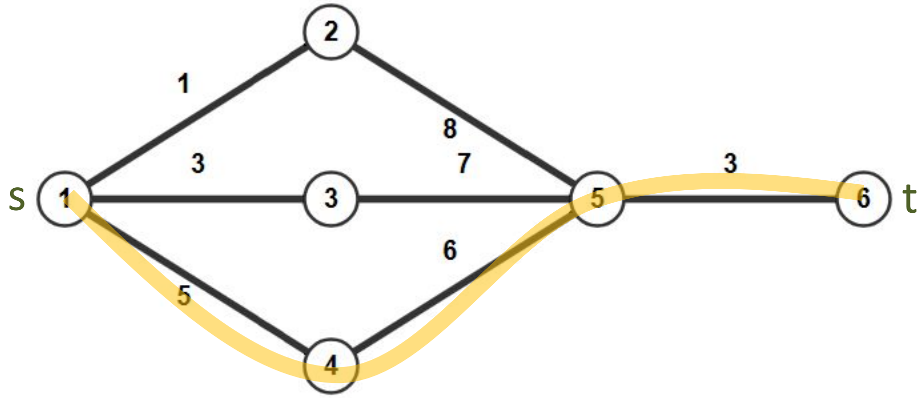
```
typedef pair<int, int> ii;
typedef pair<int, ii> iii;

int Kruskal(vector<iii>& edges, int n) {
    sort(edges.begin(), edges.end());
    unionfind components(n);
    int mst_cost = 0;
    for (iii e : edges) {
        if (components.find(e.second.first)
                    != components.find(e.second.second)) {
            components.unite(e.second.first, e.second.second);
            mst_cost += e.first; }}
    return mst_cost;
}
```

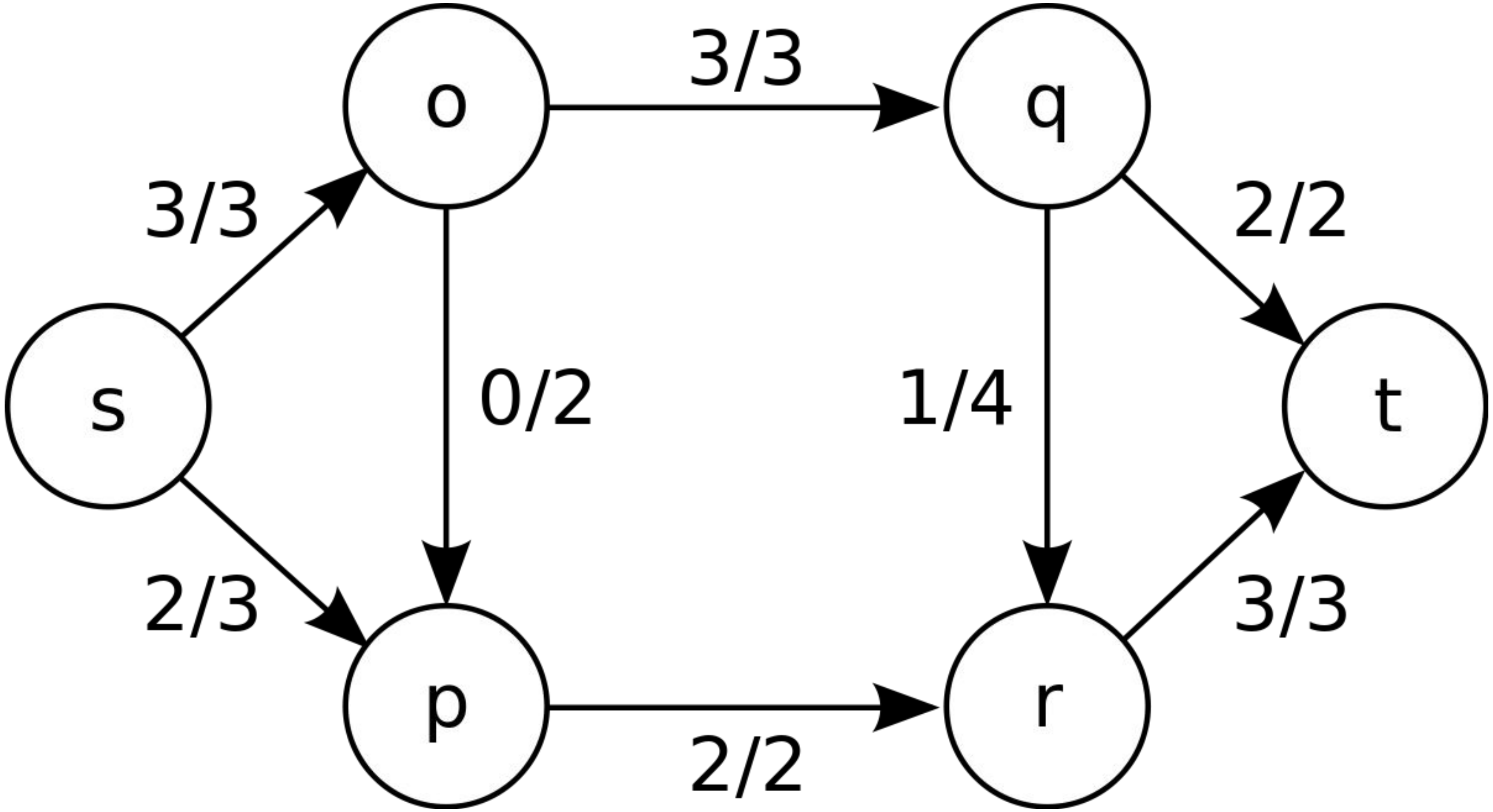
Know this! Sometimes problems are variants that require changing this code.

Minimax Paths

- Input: Weighted undirected graph, source and destination.
- Output: A path such the weight of the heaviest edge is minimal.



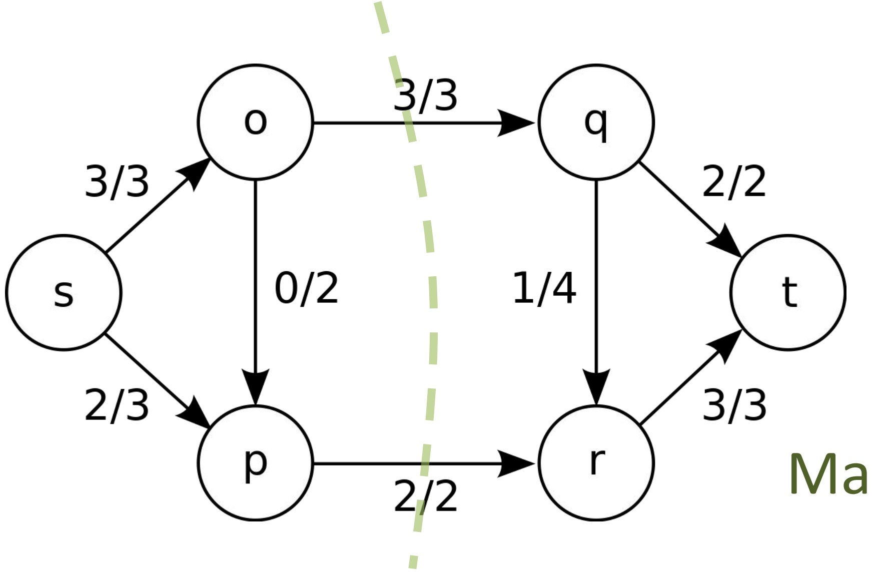
- This is the path between the two nodes in a minimum spanning tree.
  - To find it, **compute the MST and take the path between the two nodes**.
- The opposite (maximizing the lightest edge) is called Widest Path.
  - The widest path lies on a maximum spanning tree.



Max Flow

Max Flow

- Input: Directed graph with source and target, capacities on edges
- Output: The max possible flow from s to t



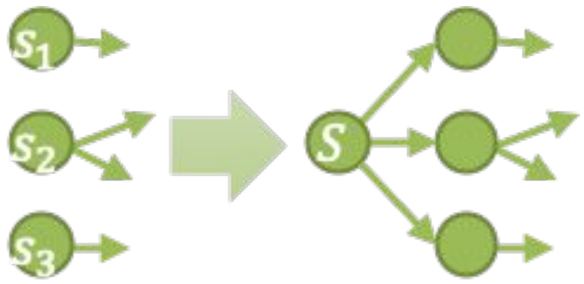
Max Flow = 5

- Property:
  "Min Cut – Max flow": max flow = min capacity of an s-t cut

Max Flow: Variants Simulation

- Vertex capacity can be simulated by splitting the vertex to two, and adding an edge between them with the capacity

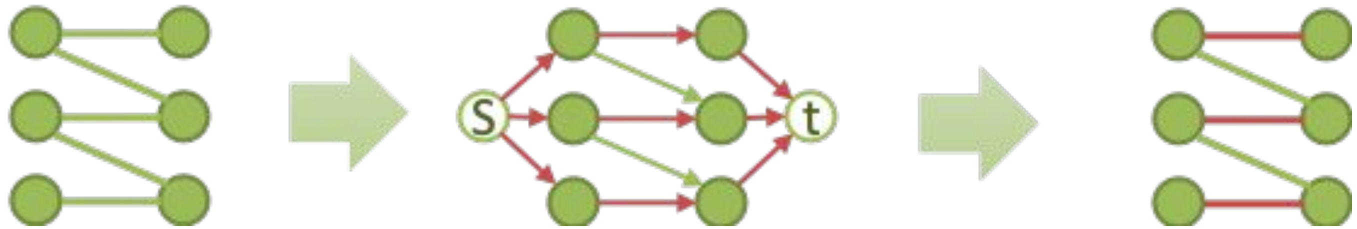
- Multiple sources can be simulated by adding a single source with outgoing edges to all sources (same for multiple targets)



Max Flow: Application 1

- Problem: Find a maximum matching on a bipartite graph
  (max number of edges to keep s.t. each vertex touches at most 1 edge)

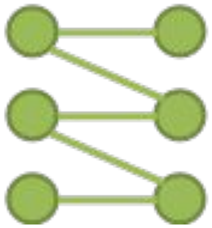
- Solution:
  - Build flow network
  - Connect one side to s and the other side to t
  - Use unit weights on edges



Max Flow: Application 1

- Problem: Find a maximum matching on a bipartite graph
  (max number of edges to keep s.t. each vertex touches at most 1 edge)

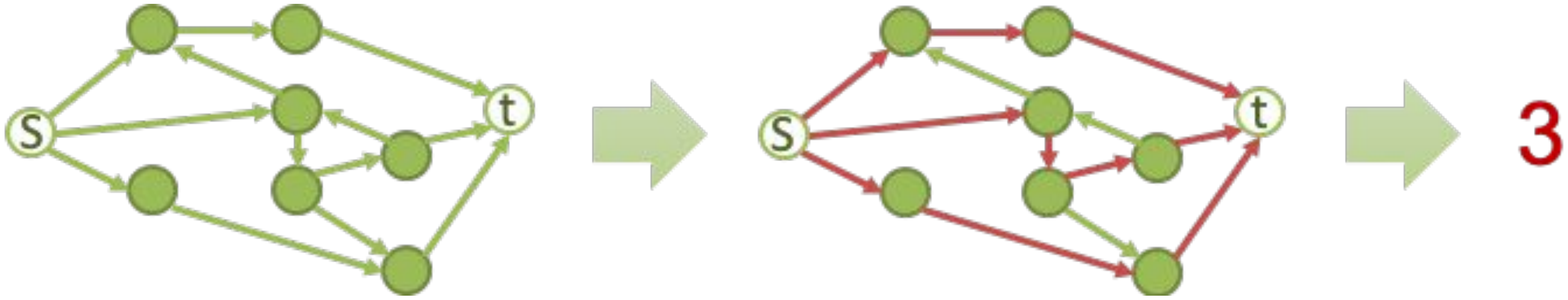
- Properties in bipartite graphs:
  - Maximum matching = Min vertex cover
    (min number of nodes that touch all edges)
  - Maximum matching = V – Max Independent Set
    (max number of nodes that do not share an edge)



Max Flow: Application 2

- Problem: Find maximum s-t edge-disjoint paths
  (max number of s-t paths s.t. each edge appears in at most 1 path)

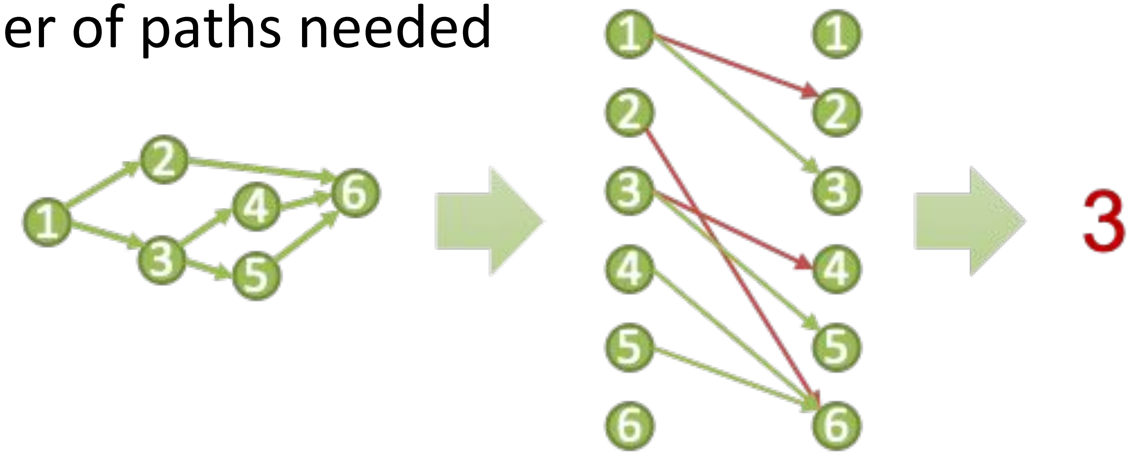
- Solution:
  - Treat graph as flow network
  - Assign each edge with unit capacity



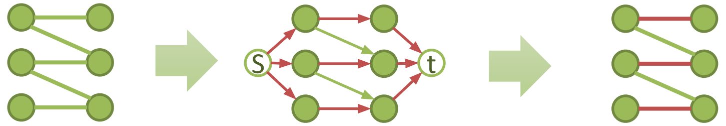
Max Flow: Application 3

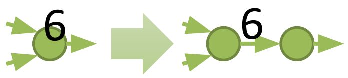
- Problem: Find a minimum path cover on a DAG
  (the min number of paths to cover the vertices in vertex-disjoint paths)

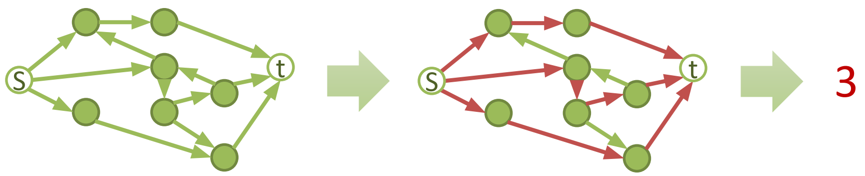
- Solution:
  - Duplicate each node and direct edges from left side to right
  - Find max matching on the bipartite graph
  - |V| - Size of max matching = Number of paths needed

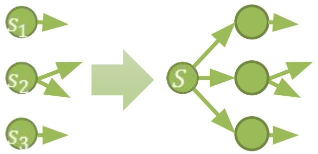


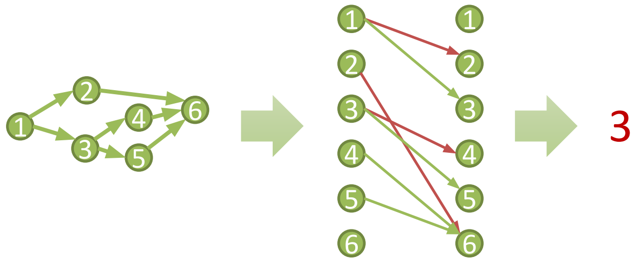
Original graphs (for editing purposes):











Max Flow: Algorithm

- Given a flow, the residual network holds the possible changes in flow. Given $u \to v$ with capacity c and flow f, the residual network has:
  - An edge $u \to v$ with capacity $c - f$
  - An edge $v \to u$ with capacity f

- Ford-Fulkerson: As long as there is an s-t path in the residual graph, send flow along one such path.
  - Such a path is called an augmenting path.

Finding Max Flow

	Ford-Fulkerson	Edmonds–Karp	Dinitz
Time			
Type	General method.	Specific version of Ford-Fulkerson.	Makes the same choices as Edmonds-Karp, but more efficient.
Description	Build the residual graph. As long as you can, send flow along an augmenting path.	Choose a shortest augmenting path at each step (Use BFS from s to find a shortest path).	Improves on Edmonds-Karp by using some data structure, but harder to implement.

F* = max flow

Implementation

BFS
Only on edges with residual
Save the BFS tree

```
int addedFlow, maxFlow = 0;
do {

    vi dist(res.size(), INF); dist[s] = 0;
    queue<int> q; q.push(s);
    vi p(res.size(), -1);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;
      for (int v : adj[u]) if (res[u][v] > 0 && dist[v] == INF) {
        dist[v] = dist[u] + 1;
        q.push(v);
        p[v] = u; }}

    addedFlow = augment(res, s, t, p, INF);
    maxFlow += addedFlow;

} while (addedFlow > 0);
```

Augment path

Implementation

Go backwards from t to s according to p

```
int augment(vvi& res, int s, int t, const vi& p, int minEdge) {

    if (t == s) {
        return minEdge;
    } else if (p[t] != -1) {

        int f = augment(res, s, p[t], p, min(minEdge, res[p[t]][t]));

        res[p[t]][t] -= f;
        res[t][p[t]] += f;

        return f;}
    return 0;
}
```

Going in: find the min edge weight on the path
Going out: update all edges with this weight