Maximum Flow

We know that graphs can be used to model problems involving traversal and routes, like the shortest path problem. Another useful interpretation of a directed graph is as a **flow network**, and solve problems about material flows. We consider a source node and a sink node, and analyse the flow of a material through the network from the source to the sink. Flow networks can model many problems, such as liquid flows through pipes, parts through assembly lines, electricity through wires, etc.

Flow Networks

A **flow network** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a non negative **capacity** $c(u, v) \geq 0$. If $(u, v) \notin E, c(u, v) = 0$.

We have two special vertices in a flow network, a **source** s and a **sink** t. We assume that there are no incoming edges to s, and no outgoing edges from t.

For convenience, we assume that all vertices v lie on a path from s to t. So, $s \rightsquigarrow v \rightsquigarrow t$ is a path in the graph. Thus the graph is connected, and $|E| \geq |V| - 1$.

We also assume, that there are no self loops in the graph and no loops of size 2 (antiparallel edges), i.e. if $(u, v) \in E$, then $(v, u) \notin E$.

A **flow** in G is a real valued function $f : V \times V \to \mathbb{R}$ that satisfies the following properties:

1. **Capacity constraint:** $\forall u, v \in V$ we require $f(u, v) \leq c(u, v)$.
2. **Skew symmetry:** $\forall u, v \in V$ we require $f(u, v) = -f(v, u)$.
3. **Flow conservation:** For all $u \in V \setminus \{s, t\}$, we require

$$\sum_{v \in V} f(u, v) = 0 = \sum_{v \in V} f(v, u)$$

The value $f(u, v)$ (which can be zero, positive or negative) is called the **flow** from u to v.

The **value** of flow f is defined as

$$|f| = \sum_{v \in V} f(s, v)$$

i.e. the total flow out of the source.

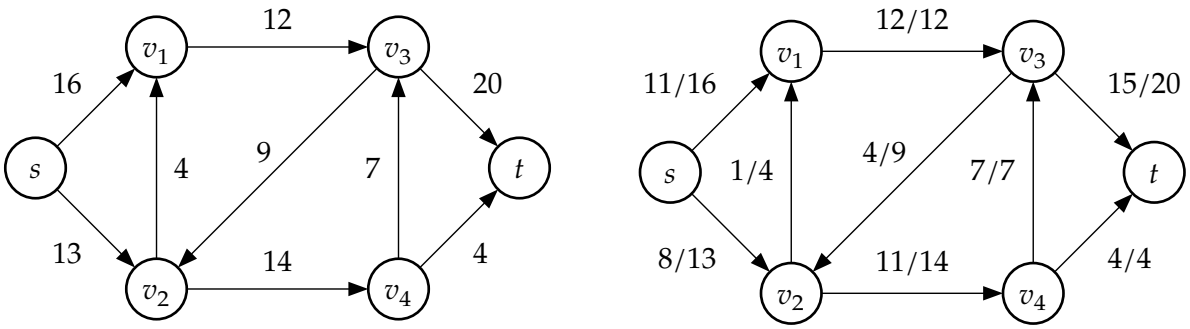

Figure 1: A flow network G, and a flow f on G with value $|f| = 19$. The notation a/b on an edge (u, v) denotes $f(u, v) = a$ and $c(u, v) = b$. Only positive flow values are shown.

- The capacity constraint restricts the flow through an edge to exceed its capacity.

- Skew symmetry is a notational convenience which says that flow from u to v is the negative of the flow from v to u.
- Flow conservation states that total incoming flow (or total outgoing flow) in a non sink-source should be 0.
- When neither (u, v) nor (v, u) exists in E, then $f(u, v) = f(v, u) = 0$.

We also define the **total positive flow entering** v as

$$\sum_{\substack{u \in V \\ f(u,v)>0}} f(u, v)$$

and similarly, the **total positive flow exiting** v. The **total net flow** at v is the total positive flow minus the total negative flow. So, flow conservation states, for any non source/sink node, its total net flow must equal 0.

A notational convenience

We extend the definition for f for sets of vertices, as follows:

$$f(X, y) = \sum_{x \in X} f(x, y) \qquad f(x, Y) = \sum_{y \in Y} f(x, y) \qquad f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$$

Then, flow conservation can be expressed simply as $f(u, V) = f(V, u) = 0$ for all $u \in V \setminus \{s, t\}$.

The value of a flow is $|f| = f(s, V)$.

Some Identities

Let $G = (V, E)$ be a flow network, and let f be a flow in G. Then,

1. For all $X \subseteq V, f(X, X) = 0$.

   Proof: $f(X, X) = \sum_{u,v \in X} f(u, v) = \sum_{u,v \in X} -f(v, u) = -f(X, X) \Rightarrow f(X, X) = 0$.

2. For all $X, Y \subseteq V, f(X, Y) = -f(Y, X)$.

   Proof: $f(X, Y) = \sum_{u \in X} \sum_{v \in Y} f(u, v) = \sum_{u \in X} \sum_{v \in Y} -f(v, u) = -f(Y, X)$.

3. For all $X, Y, Z \subseteq V$, $X \cap Y = \phi$, we have

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$$
$$f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$$

   Proof:

$$f(X \cup Y, Z) = \sum_{u \in X \cup Y} f(u, Z) = \sum_{u \in X} f(u, Z) + \sum_{u \in Y} f(u, Z) = f(X, Z) + f(Y, Z)$$
$$f(Z, X \cup Y) = \sum_{u \in X \cup Y} f(Z, u) = \sum_{u \in X} f(Z, u) + \sum_{u \in Y} f(Z, u) = f(Z, X) + f(Z, Y)$$

Using these identities, we can show e.g. $|f| = f(V, t)$.

$$\begin{aligned}|f| &= f(s,V)\\ &= f(V,V) - f(V-s,V)\\ &= f(V,V-s)\\ &= f(V,t) + f(V,V-\{s,t\})\\ &= f(V,t) - \sum_{u\in V\setminus\{s,t\}} f(u,V)\\ &= f(V,t)\end{aligned}$$

Maximum Flow Problem: Given a flow network G with source s, sink t and capacity function c, find a flow of maximum value.

How might one approach solving this problem? A naive approach might be to find some path from s to t in which all edges are edges of G, and all these edges can admit more flow, and keep increasing the flow, until no such path exists. This algorithm doesn't always terminate on the maximum flow. (Come up with a counter example.)

Max-Flow Min-Cut Theorem

A **cut** (S,T) of a flow network G is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.

If f is a flow, then the **net flow** across the cut (S,T) (with respect to f) is defined as $f(S,T)$.

The **capacity** of the cut is defined as $c(S,T)$. A **minimum cut** of a network is a cut whose capacity is the minimum over all possible cuts of the network.

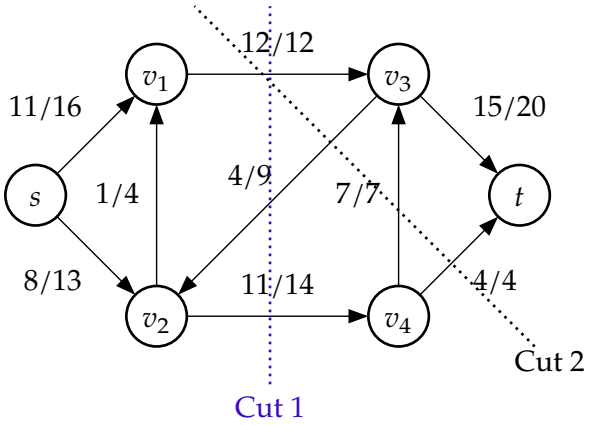

Figure 2: Two cuts (S_1,T_1) and (S_2,T_2) of G. $S_1 = \{s,v_1,v_2\}$. $S_2 = \{s,v_1,v_2,v_4\}$

The above figure shows two possible cuts of G.

The net flow across Cut 1 is

$$\begin{aligned}f(S_1,T_1) &= f(v_1,v_3) + f(v_2,v_3) + f(v_2,v_4)\\ &= 12 + (-4) + 11 = 19\end{aligned}$$

and the capacity of Cut 1 is

$$\begin{aligned}c(S_1,T_1) &= c(v_1,v_3) + c(v_2,v_4)\\ &= 12 + 14 = 26\end{aligned}$$

Similarly, the net flow across Cut 2 is

$$\begin{aligned} f(S_2, T_2) &= f(v_1, v_3) + f(v_2, v_3) + f(v_4, v_3) + f(v_4, t) \\ &= 12 + (-4) + 7 + 4 = 19 \end{aligned}$$

and its capacity is

$$\begin{aligned} c(S_2, T_2) &= c(v_1, v_3) + c(v_4, v_3) + c(v_4, t) \\ &= 12 + 7 + 4 = 23 \end{aligned}$$

Observe that the net flow across both the cuts is 19, which is also the value of the flow f. This is not a coincidence.

Lemma 1: Let f be a flow in G with source s and sink t. Let (S, T) be a cut in G. Then the net flow across (S, T) is $f(S, T) = |f|$.

Proof:

$$\begin{aligned} f(S, T) &= f(S, V) - f(S, S) \\ &= f(S, V) \\ &= f(s, V) + f(S - s, V) \\ &= f(s, V) \\ &= |f| \end{aligned}$$

In other words, given a network G with a flow f, the flow across any cut (S, T) will be equal to the value of the flow $|f|$.

From our capacity constraint, we know that $f(u, v) \leq c(u, v)$ for all $(u, v) \in V \times V$. So from this, and Lemma 1, we can write:

Lemma 2: The value of any flow f in G is bounded above by the capacity of any cut of G.

Proof:

$$\begin{aligned} |f| &= f(S, T) \\ &= \sum_{\substack{u \in S \\ v \in T}} f(u, v) \leq \sum_{\substack{u \in S \\ v \in T}} c(u, v) \\ &= c(S, T) \end{aligned}$$

In other words, the value of a flow can never exceed the capacity of any cut of the network.

As a consequence, the **maximum flow in a network is bounded above by the minimum cut of the network**. The value of the flow is the total flow passing through the network, so it must also pass through the minimum cut. The **minimum cut** is the bottleneck which bounds the possible maximum flow rate of the material through the network.

We state a theorem that states that in fact both these quantities are equal. We shall be proving this theorem shortly.

> **Max-Flow Min-Cut Theorem**
>
> For any flow network $G = (V, E)$ with source s and sink t, the maximum flow equals the minimum cut capacity.

Residual Graph and Augmenting Paths

Let's think of how we can try to find the maximum flow of the network. Given a flow, can we check if we can increase its value? That is, could we somehow push more material through the network?

Consider a pair of vertices (u, v). By capacity constraint, $f(u, v) \leq c(u, v)$. So, intuitively, the additional flow we can 'push' through (u, v) is $c(u, v) - f(u, v)$ without violating the constraints.

Define the **residual capacity** of a pair of vertices $(u, v) \in V \times V$ as:

$$c_f(u, v) = c(u, v) - f(u, v)$$

Notice that this also holds if $c(u, v) = 0$ and $f(u, v)$ is negative. This can happen if $(v, u) \in E$. Consider, $f(u, v) = -5, c(u, v) = 0$. Then, we can 'increase' the flow from (u, v) by 5 units, which is mathematically equivalent to decreasing the flow from v to u to 0.

Define the **residual graph** of G induced by flow f as $G_f = \left(V, E_f\right)$, where

$$E_f = \left\{(u, v) \in V \times V : c_f(u, v) > 0\right\}$$

Essentially, the residual graph consists of edges that can admit more flow.

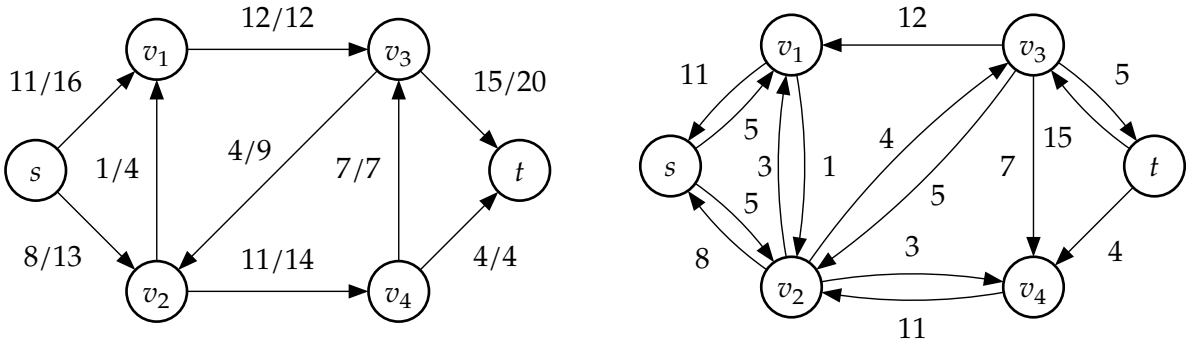

Figure 3: Residual graph of G induced by f.

Let's call a edge $(u, v) \in E$ to be **saturated**, if $f(u, v) = c(u, v)$, i.e. no more flow can be pushed through the edge. Call an edge to be **zeroed out** if $f(u, v) = 0$.

Some interesting properties of the residual graph (which can be shown):

1. If an edge $(u, v) \in E$, if (u, v) is saturated, then $(v, u) \in E_f$, but $(u, v) \notin E_f$.
2. If an edge $(u, v) \in E$, if (u, v) is zeroed out, then $(u, v) \in E_f$, but $(v, u) \notin E_f$.
3. If an edge $(u, v) \in E$ is neither saturated nor zeroed out, then $(u, v) \in E_f$ and $(v, u) \in E_f$.

Each edge of the residual network is a **residual edge** which can admit more flow.

We need to find the maximum flow from s to t. For this, we look at paths from s to t in G_f. Given a flow network $G = (V, E)$ and a flow f, an **augmenting path** p is a simple path from s to t in G_f.

Clearly, every edge on a augmenting path can admit more flow. This means, that we have a potential opportunity to increase the total flow from s to t.

Define the **residual capacity** of path p as

$$c_f(p) = \min\{c_f(u,v) \mid (u,v) \in p\}$$

Clearly, we can add $c_f(p)$ amount of flow to each edge in p, and still have a valid flow.

We define a procedure AUGMENT on a augmenting path p, which updates the flow f by adding $c_f(p)$ to the flow of all edges in path p.

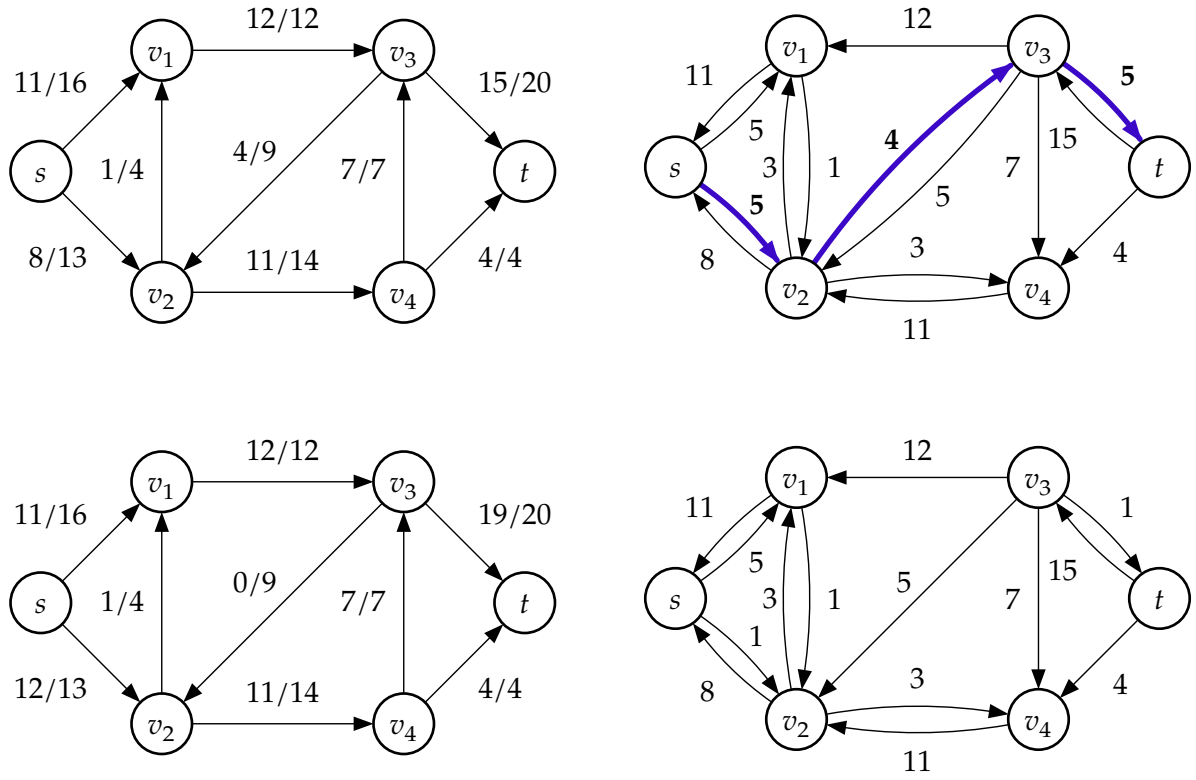

Figure 4: Augmenting flow f along a path p. (a) A flow f. (b) An augmenting path p in G_f. (c) The augmented flow f'. (d) The residual graph $G_{f'}$ after updating the flow f.

In the above example, we start with a flow f on G. In the residual graph G_f, we find a augmenting path $p = \langle s, v_2, v_3, t \rangle$. Its residual capacity is the minimum residual capacity of its edges, which is $c_f(p) = \min(5, 4, 5) = 4$.

This indicates, we can push 4 units of material through path p, which should increase the value of flow f by 4. The updated flow values are shown in (c).

Note that the edge (v_3, v_2) originally had flow 4 before augmentation, but has 0 flow after. Since the path p moves from v_2 to v_3, the flow value on the real edge (v_3, v_2) *decreases*. This matches our notion of skew-symmetric flows, mathematically increasing flow on a reverse edge (v, u) is equivalent to decreasing flow on the real edge (u, v). (Unlike the naive algorithm, where once a flow has been set it can never decrease, using the skew-symmetric flow function and the residual graph allows us to 'undo' our mistakes, i.e. reduce the flow of material through an edge, if it leads to a better flow.)

Lemma 3: (Flow Augmentation) Let f be a flow in G, and let p be an augmenting path from s to t in G_f. Define the **augmented flow** f' as

$$f'(u,v) = \begin{cases} f(u,v) + c_f(p) & \text{if } (u,v) \in p \\ f(u,v) - c_f(p) & \text{if } (v,u) \in p \\ f(u,v) & \text{otherwise} \end{cases}$$

Then, f' is a flow in G, and $|f'| > |f|$.

Proof:

1. Capacity constraint: If $(u,v) \in p, f'(u,v) = f(u,v) + c_f(p) \leq f(u,v) + c_f(u,v) = c(u,v)$.
2. Flow conservation: For some node $u \in V \setminus \{s,t\}$ which lies on path p, with predecessor x and successor y. $((x,u) \in p, (u,y) \in p)$ Then

$$\begin{aligned} f'(u,V) = \sum_{v \in V} f'(u,v) &= f'(u,v_1) + \ldots + f'(u,x) + f'(u,y) + \ldots + f'(u,v_n) \\ &= f(u,v_1) + \ldots + f(u,x) + c_f(p) + f(u,y) - c_f(p) + \ldots + f'(u,v_n) \\ &= \sum_{v \in V} f(u,v) = f(u,V) = 0 \end{aligned}$$

3. Skew symmetric: Let $(u,v) \in p$, then $f'(v,u) = f(v,u) - c_f(p) = -f(u,v) - c_f(p) = -f'(u,v)$. If $(u,v) \notin p \wedge (v,u) \notin p$ then $f' = f$ so skew symmetry holds.
4. Value of flow: The value of flow has been modified for exactly one edge out of all the edges connected with s.

$$|f'| = f'(s,V) = \sum_{v \in V} f'(s,v) = \sum_{v \in V} f(s,v) + c_f(p) = |f| + c_f(p) > |f|$$

At this point, we are at a position to give the proof of the max-flow min-cut theorem: (The following is an equivalent restatement)

Theorem 1: (Optimality Conditions for Maximum Flow) If f is a flow in $G = (V,E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual graph G_f contains no augmenting paths.
3. $|f| = c(S,T)$ for some cut (S,T) in G.

Proof:

$1 \Rightarrow 2$: Let f be a maximum flow in G. Suppose, that there does exist some augmenting path p in G_f. Then, by the AUGMENT procedure (Lemma 3), we can find a new flow f', such that $|f'| > |f|$. This contradicts the maximality of f. Hence, there does not exist any augmenting path in G_f.

$2 \Rightarrow 3$: In G_f, there does not exist any paths from s to t. Then, define

$$S = \left\{ v \in V \mid v \text{ is reachable from } s \text{ in } G_f \right\}.$$

Clearly, $s \in S, t \notin S$. In the cut $(S, T = V - S)$, consider any pair of vertices (u,v), $u \in S, v \in T$. For these pairs, $c_f(u,v) = 0$ (since these aren't edges in G_f, otherwise we could reach vertices in T from s), which means that $f(u,v) = c(u,v)$. So, summing over all such pairs, we get

$$\begin{aligned} \sum_{u \in S} \sum_{v \in T} f(u,v) &= \sum_{u \in S} \sum_{v \in T} c(u,v) \\ f(S,T) &= c(S,T) \\ |f| &= c(S,T) \end{aligned}$$

$3 \Rightarrow 1$: We have, $|f| = c(S, T)$. By Lemma 2, this is necessarily a maximum flow. □

In Figure 4(d), note that there are no more augmenting paths in G_f. Thus, the flow f' is a maximum flow, and the value of the maximum flow is $f(s, V) = 23$ (which is also the capacity of cut 2 (Figure 2), which is a min cut!)

Ford Fulkerson Algorithm

We have shown that the maximum flow is equal to the minimum cut in the network. Lemma 3 also gives us a way to augment a flow to get a flow of higher value. We can use it to get an algorithm for finding the maximum flow.

The basic method for finding the max-flow is, we start with a 0 flow (all $f(u, v) = 0$), choose some augmenting path p in G_f, and augment the flow by Lemma 3. Continue doing so, till there are no augmenting paths. At this point, the flow is a maximal flow, by Theorem 1.

FORD-FULKERSON-METHOD(G, s, t)

1 initialise f to 0
2 **while** there exists an augmenting path p in G_f
3 augment flow f along p
4 **return** f

See Figure 26.5 from CLRS Second Ed (or Figure 26.6 from CLRS Third Ed) for a demonstration of complete execution of this algorithm.

Here, the convergence and time complexity of this algorithm depends on the type of the graph, and how the path p is chosen on line 2.

Basic Algorithm Complexity

First, let's assume that all edge capacities are integers, and we pick the path p arbitrarily (say, using DFS). We will show that the algorithm runs in $O(E\,|f^*|)$ time, where f^* is a max-flow.

Each iteration of the algorithm requires finding a path p, which takes $O(V + E) = O(E)$, and augmenting the flow along path p, which is also $O(E)$. Initially, the value of flow f is 0, and after each iteration, it increases by at least 1 unit (since all edges have integral capacities). So in at most $|f^*|$ iterations, the algorithm will find the max flow and then terminate. Hence the algorithm's time complexity is $O(E|f^*|)$.

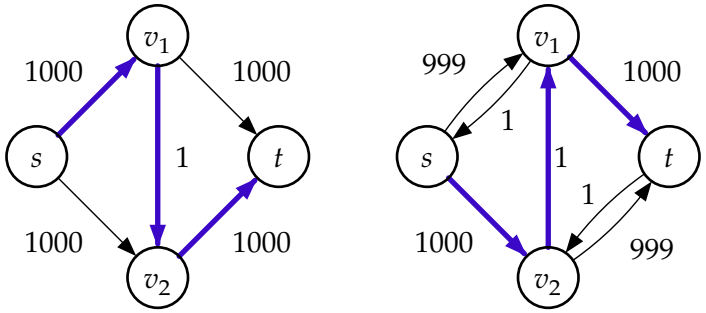

Figure 5: A flow network for which Ford-Fulkerson may takes $O(E\,|f^*|)$ time.

In the above example, the value of maximum flow is $|f^*| = 2000$. But suppose our choice of path p is such that we choose $p_1 = \langle s, v_1, v_2, t\rangle$ in the first iteration, then $c_f(p_1) = 1$, and we increase our flow by 1. Then, our choice of path might be $p_2 = \langle s, v_2, v_1, t\rangle$, in which case again, $c_f(p_2) =$

1 and our flow increases by 1. Assuming we keep choosing p_1 in odd iterations and p_2 in even iterations, to reach the maximum flow of 2000, the algorithm will need 2000 iterations.

Also, since $|f^*| \leq \sum_{e \in E} c(e)$, we can write the time complexity as $O(|E| \sum c(u,v))$.

Observe, that the time complexity of the algorithm is a polynomial function of the *input values* (not of the **input size**!). Such a complexity is said to be **pseudo-polynomial** time.[1]

Edmonds-Karp Algorithm

Can we do better, and get a algorithm which runs in running time polynomial in the input size?

The bound on Ford-Fulkerson can be improved if we implement the choice of path p using a Breadth-First Search, i.e. if we always select a shortest path (in terms of number of edges) from s to t in G_f. This algorithm is called the **Edmonds-Karp algorithm**. It can be shown to run in $O(m^2 n)$ time.

EDMONDS-KARP(G, s, t)

1 initialise f to 0
2 **while** there exists an augmenting path in G_f
3 find a shortest augmenting path p by running BFS on G_f
4 augment flow f along p
5 **return** f

(Note that for purposes of defining 'a shortest path' and 'length of a path', we consider graphs to be unweighted, so each edge contributes to a length of 1 unit.)

We will prove the time complexity bound for this algorithm to be $O(m^2 n)$. First, the body of the while loop runs in $O(m + n) = O(m)$ (BFS and updating the flow). It suffices to show, that the number of iterations that the while loop runs for is $O(mn)$.

Layered Residual Graph

For the purposes of this proof, let us define the **layered residual graph** G_f^L, which is a subgraph of G_f as follows:

- $\delta_G(s,v)$ = length of the shortest path from s to v in G.
- Let $L_i = \left\{v \in V \mid \delta_{G_f}(s,v) = i\right\}$, the set of vertices whose shortest path distance from s is i.
- Let $\delta_{G_f}(s,t) = \ell$.
- $G_f^L = \left(V, E_f^L\right)$, $E_f^L = \{(u,v) \in E \mid u \in L_i, v \in L_{i+1} \text{ for some } i\}$.

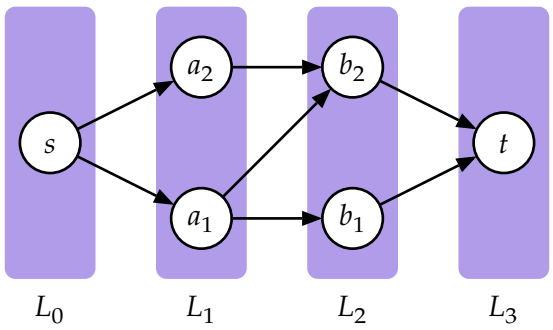

Figure 6: Structure of a layered residual graph

[1] In terms of the input size, this algorithm runs in exponential time.

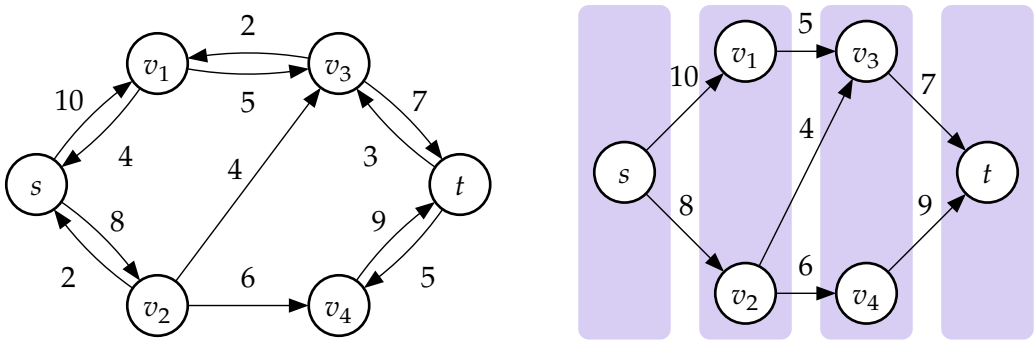

Figure 7: A residual graph G_f and its layered residual graph G_f^L

In other words, in the layered residual graph, we have arranged the vertices into layers according to the shortest path distances from s. We only keep the edges which go from a vertex closer to s to a vertex farther from s. We remove any **back edges** (edges that go to a node which is closer to s), and **cross edges** (edges that connect nodes whose distance from s is the same).

Lemma 4: Any shortest path from s to v in G_f also exists in G_f^L.

Proof: Let $v \in L_i$, let p be a shortest path from s to v in G_f. Assume, that p does not exist in G_f^L. For this to happen, p must contain some edge which was removed, so one of the edges of p must be a back edge or a cross edge. Now, since $v \in L_i$, and $s \in L_0$, and at least one edge in p is a back edge or a cross edge, this implies that $|p| > i$. But this contradicts the minimality of p. Thus, any shortest path from s to v in G_f must also be present in G_f^L.

This lemma essentially states, that **the layered residual graph G_f^L is a succinct representation of the shortest paths in G_f.**

Complexity

Now we can come to the analysis of the number of iterations of the algorithm.

Assume that EDMONDS-KARP runs for k iterations.

After the ith iteration, let the flow be f_i. The corresponding residual graph is G_{f_i}.

Define $\delta_i = \delta_{G_{f_i}}(s,t)$. We have, $\delta_k = \infty$ (after k iterations, s and t are disconnected.)

Claim: $\delta_0 \leq \delta_1 \leq \delta_2 \leq \cdots \leq \delta_{k-1} < \delta_k = \infty$.

Proof: Let the current distance between s and t be δ_i. In the next iteration, we choose a shortest path from s to t (in G_{f_i}), and augment our flow along this path.

Call an edge (u,v) in an augmenting path p in G_f **critical**, if $c_f(p) = c_f(u,v)$, i.e. it's residual capacity is the minimum on its path. Each augmenting edge has at least one critical edge. When we augment the flow along p, (u,v) gets saturated.

Once we choose some shortest path p in G_{f_i}, let (u,v) be a critical edge in p. After augmentation, the resulting residual graph is $G_{f_{i+1}}$. Due to augmentation, the edge (u,v) will become saturated, so won't be present in $G_{f_{i+1}}$.

What is δ_{i+1}?

1. Case 1: There is still some path p in $G_{f_{i+1}}$ of length δ_i.

   Then, $\delta_{i+1} = \delta_i$.

2. Case 2: Otherwise: $\delta_{i+1} > \delta_i$. This is because, if δ_{i+1} were to be smaller than δ_i, that would mean there is a path from s to t in $G_{f_{i+1}}$ shorter than δ_i. But this path must then also exist in G_{f_i}. This is a contradiction by definition of δ_i.

So we have shown, $\delta_{i+1} \geq \delta_i$.

We can also bound the value of δ_i after any iteration i: $\delta_i \leq n - 1$, since length of any path in a n node graph cannot exceed $n - 1$.

Now, to get a bound on the total number of iterations (k), we need to bound the number of iterations for which δ_i can remain unchanged.

Let

$$\delta_{i-1} < \delta_i = \delta_{i+1} = \delta_{i+2} = \ldots = \delta_{i+t} < \delta_{i+t+1}$$

What is the maximum value of t? Observe, that after each iteration, at least one edge gets removed from G_{f_i}. Importantly, G_f^L only loses edges, it never gains new edges. This is because the critical edge is always in G_f^L (by Lemma 4) which it loses, and the possible edges that get added to G_f are the reverse edges of those in p, which are always back edges, so they don't exist in G_f^L.

Then, after at most $|E|$ iterations, there will not exist any more shortest paths of length δ_i. Therefore, $t \leq m$.

Overall, this bounds the total number of iterations to be $k \leq mn$ (n possible values of δ_i, and at most m iterations spent for each value).

Thus, as each iteration takes $O(m)$ time, and there are at most mn iterations, EDMONDS-KARP runs in $O(m^2 n)$ time, which is polynomial in the input.

Dinic's Algorithm

We can try to do even better. Dinic's algorithm is a modification on EDMONDS-KARP, which runs in $O(n^2 m)$ time.

First, let's identify, can we decrease the number of iterations? If we could ensure $\delta_{i+1} < \delta_i$ (strict inequality), then our number of iterations of our algorithm is bounded by n. For this, we need Case 1 (above) to not happen. So, we must eliminate all shortest paths of length δ_i in the ith iteration itself.

So, in a single iteration, we find a set of $s - t$ paths in the layered residual graph G_f^L, till at least one edge on every $s - t$ path has been saturated; such a set of paths forms a **blocking flow**. Then, we augment our flow with this blocking flow.

DINIC(G, s, t)

```
1  initialise f to 0
2  while there exists an augmenting path in G_f
3      build the layered residual graph G_f^L
4      compute a blocking flow b in G_f^L
5      f ← f + b    ▷ augment f with b
6  return f
```

A flow f in a network G is a **blocking flow**, if every $s-t$ path in G contains at least one saturated edge.

We argue about the correctness of the algorithm: After each iteration we get a blocking flow b of G_f^L, and then augment f by b. Let the $s-t$ before an iteration be δ_i, then after the augmentation, it necessarily has to be $> \delta_i$, since all shortest paths of length δ_i have at least one edge removed. So there are at most n iterations. Also, since δ increases monotonically, after at most n iterations it must grow to ∞, i.e. s and t become disconnected in G_f. By Theorem 1, f is a maximum flow of G.

Since each blocking flow computation (line 4) can be done in $O(mn)$ (Problem 16), and there are at most $O(n)$ iterations, Dinic's algorithm runs in a time bound of $O(n^2m)$ which is a better asymptotic bound than Edmonds-Karp for dense graphs.

Summary

We have discussed 3 algorithms/methods for finding the Maximum Flow in a network G, summarized below:

	Augmenting flow	**Time complexity**	**Type**
FORD-FULKERSON	Arbitrary $s-t$ path	$O(m\|f\|) = O(m\sum c)$	Pseudo-Polynomial
EDMONDS-KARP	Shortest $s-t$ path	$O(nm^2)$	Polynomial
DINIC	Blocking flow	$O(n^2m)$	Polynomial

Table 1: Summary of all three max-flow algorithms

Next up, we will look at a different paradigm for solving max-flow problems, which lead to some even more optimized algorithms for computing a max-flow.

Problems

1. **[Simplifying Input Instance]** Explain how we can assume without loss of generality that the input directed graph for the network flow problem does not contain any self-loops or anti-parallel edges. For two vertices u and v, if the graph has an edge from u to v and another edge from v to u, then these pair of edges are called anti-parallel edges.

Let $G = (V, E)$ be the input directed graph. We construct an equivalent graph $G' = (V', E')$ which does not contain any self-loops or anti-parallel edges.

To construct G',
1. remove all self loops $(v, v) \in E, v \in V$ from E.
2. fix anti-parallel edges. Suppose $(v_1, v_2) \in E$ and $(v_2, v_1) \in E$. Then, remove the edge (v_2, v_1), and a new node v, and add edges (v_2, v) and (v, v_1). Set $c'(v_2, v) = c'(v, v_1) = c(v_2, v_1)$.

The resultant graph clearly does not contain any self-loops or anti-parallel edges.

For any flow f in G, there exists a corresponding flow f' in G', with $|f| = |f'|$:
- If f assigned any flow k to a self loop (v, v), remove it (set it to 0). This is still a valid flow with same value since both incoming and outgoing flow at v reduced by k.
- For any anti-parallel set of edges (v_1, v_2) and (v_2, v_1), set $f'(v_2, v) = f'(v, v_1) = f(v_2, v_1)$. This is still a valid flow with same value since capacity constraint is satisfied, and flow conservation on v is satisfied (only one incoming and one outgoing flow).
- For all other edges (u, v), set $f'(u, v) = f(u, v)$.

For any flow f' in G', there exists a corresponding flow f in G, with $|f| = |f'|$:
- For all self-loops, assign the flow to be 0.
- For our construction of (v_2, v) and (v, v_1) in case of anti-parallel edges in G, note that $f'(v_2, v) = f'(v, v_1)$ (flow conservation). Set $f(v_2, v_1) = f'(v_2, v)$. This is a valid flow with same value since capacity constraint is satisfied, and flow conservation in G matches that of G'.

From both these lemmas, it is clear that both graphs are equivalent, i.e. max flow value of G = max flow value of G'.

2. **[Fattest Augmenting Path]** Run the fattest path first (augment along the path where the maximum amount of flow can be sent from s to t) implementation of the Ford-Fulkerson method and the Edmond-Karp algorithm on large random graphs and evaluate their relative performance by varying the number of vertices, edges, and average capacities of the edges. Use your favorite programming language.

DIY

3. Modify your above code to compute a minimum capacity $s - t$ cut in a flow network.

DIY

4. [CLRS] Suppose that, in addition to edge capacities, a flow network has vertex capacities. That is each vertex v has a limit $\ell(v)$ on how much flow can pass though v. Show how to transform a flow network $G(V, E, w : V \to \mathbb{R}_{>0})$ with vertex capacities into an equivalent flow $G'(V', E', w' : E' \to \mathbb{R}_{>0})$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G. How many vertices and edges does G' have?

Let us assume that in G we also have edge capacities $w(u,v)$. If not, assign $w(u,v) = \infty$ for all $(u,v) \in E$.

For every node $v \in V$ in G,
- Add two nodes v_{in} and v_{out} in V'.
- Add an edge $(v_{\text{in}}, v_{\text{out}})$, with $w'(v_{\text{in}}, v_{\text{out}}) = w(v)$, the vertex capacity.

For every edge $(u,v) \in E$ in G,
- Add the edge $(u_{\text{out}}, v_{\text{in}})$ with $w'(u_{\text{out}}, v_{\text{in}}) = w(u,v)$, the original edge capacity of the edge in G.

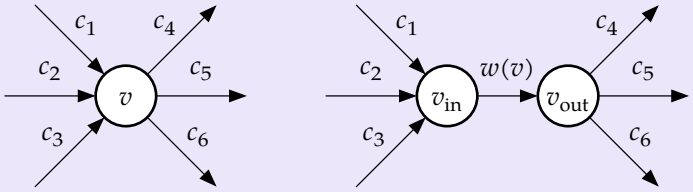

Figure 8: Transformation of a node in G into G'

Define the source and sink of G' to be s_{in} and t_{out} respectively.

G' has $2|V|$ nodes, and $|E| + |V|$ edges.

For any flow f in G (satisfying vertex and edge constraints), there exists a corresponding flow f' in G' such that $|f| = |f'|$:
- For $v \in V \setminus \{s,t\}, f'(v_{\text{in}}, v_{\text{out}}) = \sum_{\substack{u \in v \\ f(u,v)>0}} f(u,v)$, the total positive flow entering (or exiting) v.
- $f'(s_{\text{in}}, s_{\text{out}}) = f'(t_{\text{in}}, t_{\text{out}}) = |f|$, the value of the flow.
- For $(u,v) \in E, f'(u_{\text{out}}, v_{\text{in}}) = f(u,v)$.

f' is a valid flow in G', because
1. Flow conservation holds on each node v_{in} since the only outgoing edge from v_{in} has a flow equal to total positive flow entering v in G. (similarly for v_{out}, the incoming edge has flow equal to total positive flow exiting v in G).
2. Capacity constraint holds on normal edges (trivially), and on auxilliary edges due to the fact that, total positive flow entering v in G cannot exceed $w(v)$, which is the capacity of the auxilliary edge.

For any flow f' in G', there exists a corresponding flow f in G satisfying vertex and edge constraints, such that $|f| = |f'|$:
- Simply choose $f(u,v) = f'(u_{\text{out}}, v_{\text{in}})$.

This is a valid flow in G because, flow conservation is satisfied, edge capacity constraints are satisfied, and vertex capacity constraints are satisfied (as total positive flow entering v cannot exceed $w(v)$ due to an edge constraint in f').

Thus the maximum flow of G' has the same value as the maximum flow of G.

5. **[Multiple Source and Multiple Sink]** Suppose that we have multiple source and sink vertices in a flow network. The value of a flow in such a network is defined as the total amount of flow going out of all sources. Show how to transform this problem into an equivalent conventional (taught in the class) flow network such that the maximum flow value remains the same.

Let $s_1, s_2, ..., s_k$ be the source vertices and $t_1, t_2, ..., t_l$ be the sink vertices in a flow network G. We construct an equivalent network G' with a single source and sink.

Add two vertices, a **supersource** s and a **supersink** t to G'. Add edges (s, s_i) and $\left(t_j, t\right)$, all with capacity ∞. Keep all other vertices and edges same as in G.

We can convert any flow f in G into a corresponding flow f' in G', by setting the flow values of (s, s_i) as the total positive flow exiting s_i, and of $\left(t_j, t\right)$ as the total positive flow entering t_j. This is clearly a valid flow as flow conservation and capacity constraints hold. Its value is equal to the sum of flows of (s, s_i), which is the sum of total flows exiting source s_i, which is exactly the value of flow f.

Thus, these problems are equivalent.

6. **[Flow with Vertex Capacities]** Suppose every vertex v, except s, t, has a capacity c_v, meaning that at most c_v units of flow may pass through v. Design an algorithm that computes a maximum $s - t$ flow subject to both edge capacities and vertex capacities.

Convert the network to an equivalent network with only edge capacity constraints (as in Problem 4). Run a standard max-flow algorithm (Ford-Fulkerson / Edmonds-Karp). Map the max-flow in G' back to G. This gives a max $s - t$ flow in G.

7. **[Maximum Flow with Lower Bounds]** Each edge e has a lower bound l_e and upper bound u_e, and every feasible flow must satisfy $l_e \leq f_e \leq u_e$. Design an algorithm to determine whether a feasible $s - t$ flow exists. Then extend your algorithm to compute a maximum feasible s-t flow.

To determine if a feasible flow exists, we transform the network G into an equivalent max-flow problem with edge capacities G':

1. Add a supersource s' and a supersink t'.
2. For each edge $(u, v) \in E$, set its new capacity $c'(u, v) = u_{u,v} - l_{u,v}$.
3. By forcing $l_{u,v}$ flow across each edge, we create flow imbalances at the nodes (flow conservation is violated). For each vertex $v \in V$, define the **net demand**
$$\Delta(v) = \sum_{(v,w)\in E} l_{v,w} - \sum_{(u,v)\in E} l_{u,v}$$
4. If $\Delta(v) > 0$, add an edge (v, t') with capacity $\Delta(v)$. If $\Delta(v) < 0$, add an edge (s', v) with capacity $-\Delta(v)$.
5. To satisfy the conservation of flow from s to t, add an artificial edge (t, s) with infinite capacity $c'(t, s) = \infty$.

Compute the maximum $s' - t'$ flow in this new network. A feasible flow exists in the original network if and only if all edges leaving s' and entering t' are fully saturated. If so, the feasible flow on each original edge is $f(u, v) = f'(u, v) + l_{u,v}$.

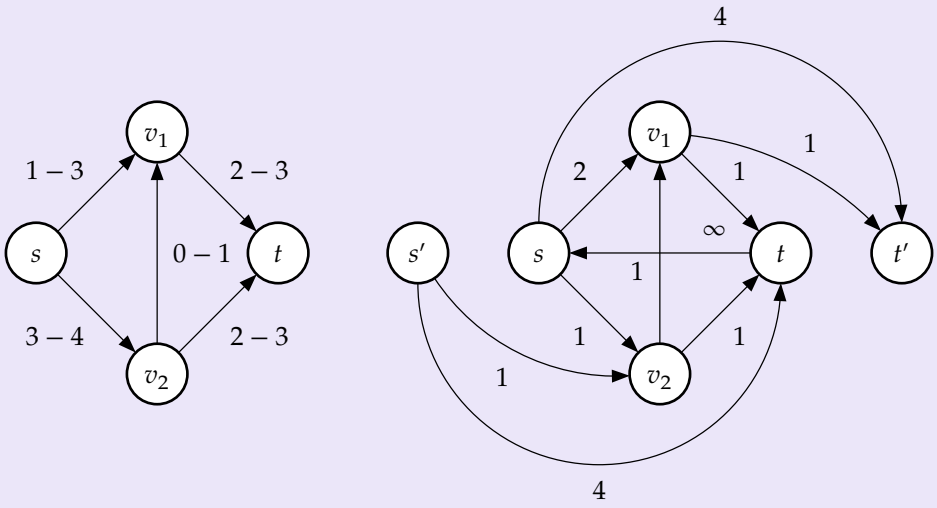

Figure 9: (a) Network G with lower and upper bounds on edges (b) Equivalent network G' with edge capacities

Proof:

1. If a feasible flow f exists in G, then a max-flow exists in G' where all s'/t' edges are saturated.

We define a flow f' in G':
- $f'(u,v) = f(u,v) - l_{u,v}, f'(t,s) = |f|, f'(s',v) = c'(s',v), f'(v,t') = c'(v,t')$.
- This is a valid flow because
  - $l_{u,v} \leq f(u,v) \leq u_{u,v} \Rightarrow 0 \leq f'(u,v) \leq c'(u,v)$
  - $v \in S - \{s,t\}$,

$$\sum_{(v,w)\in E} f(v,w) - \sum_{(u,v)\in E} f(u,v) = 0$$
$$\Rightarrow \sum_{(v,w)\in E} \left(f'(v,w) + l_{v,w}\right) - \sum_{(u,v)\in E} \left(f'(u,v) + l_{u,v}\right) = 0$$
$$\Rightarrow \sum_{(v,w)\in E} f'(v,w) - \sum_{(u,v)\in E} f'(u,v) + \sum_{(v,w)\in E} l_{v,w} - \sum_{(u,v)\in E} l_{u,v} = 0$$
$$\Rightarrow \sum_{(v,w)\in E} f'(v,w) - \sum_{(u,v)\in E} f'(u,v) + \Delta(v) = 0$$

So flow conservation holds (in either case $\Delta > 0$ or $\Delta < 0$).
  - For $v = s$ and $v = t$, flow conservation holds as total flow exiting $s = |f|$ which is in its incoming edge (t,s), and vice versa.

2. If a flow exists in G' where all s'/t' edges are saturated, then a feasible flow exists in G. (DIY :))

To compute the **maximum** feasible $s-t$ flow:
1. Find a feasible flow f of G using the above method. If no feasible flow exists, terminate.
2. Construct the residual graph G_f using the feasible flow f. The residual capacities are bounded by both the upper and lower limits:
   - Forward edge: $c_f(u,v) = u_{u,v} - f(u,v)$
   - Backward edge: $c_f(v,u) = f(u,v) - l_{u,v}$
3. Run a standard max-flow algorithm from the original s to t on G_f to augment f to f' till you reach a max-flow.

8. **[Minimum Cut After Maximum Flow]** Suppose a maximum flow has already been computed. Design an $O(|V| + |E|)$-time algorithm that finds an $s-t$ minimum cut.

Compute the residual graph. Run DFS on the residual graph G_f starting from vertex s, and find the set

$$S = \left\{ v \in V \mid v \text{ is reachable from } s \text{ in } G_f \right\}$$

Then, we claim $(S, T = V \setminus S)$ is a $s - t$ minimum cut.

This algorithm runs $O(V + E)$ time.

Firstly, note that $t \notin S$. If it were, then there would exist an augmenting path in G_f, which by Lemma 3 would imply that f is not a maximum flow. So, (S, T) is a $s - t$ cut.

Let $E^{\rightarrow}$ be all edges crossing the cut *from S to T*, and $E^{\leftarrow}$ be edges crossing the cut *from T to S*.

$$E^{\rightarrow} = \{(u, v) \in E \mid u \in S, v \in T\}$$
$$E^{\leftarrow} = \{(u, v) \in E \mid u \in T, v \in S\}$$

Then,

1. For some $(u, v) \in E^{\rightarrow}$, note that $(u, v) \notin E_f$. (If it were, then v would be reachable from s in G_f, contradicting $v \in T$). This means, $c_f(u, v) = 0 \Rightarrow f(u, v) = c(u, v)$, thus (u, v) is saturated.
2. For some $(u, v) \in E^{\leftarrow}$, note that $(v, u) \notin E_f$. This means, $c_f(v, u) = 0 \Rightarrow f(v, u) = c(v, u) = 0$, thus (u, v) is zeroed out.

The flow of the cut is

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) = \sum_{(u,v) \in E^{\rightarrow}} f(u, v) + \sum_{(u,v) \in E^{\leftarrow}} f(v, u) = \sum_{(u,v) \in E^{\rightarrow}} c(u, v) = \sum_{u \in S} \sum_{v \in T} c(u, v) = c(S, T)$$

So, we have $|f| = c(S, T)$. By Lemma 2, this implies that (S, T) is a minimum cut of G.

9. [CLRS] Suppose that you wish to find, among all minimum cuts in a flow network G with integral capacities, one that contains the smallest number of edges. Show how to modify the capacities of G to create a new flow network G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G.

Transform the edge capacities: For $(u, v) \in E$, $c'(u, v) = (m + 1)c(u, v) + 1$, where $m = |E|$.

For any cut (S, T) in G, its capacity is $c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$.

The capacity of the corresponding cut in G' is

$$c'(S, T) = \sum_{u \in S} \sum_{v \in T} c'(u, v) = \sum_{u \in S} \sum_{v \in T} (m + 1)c(u, v) + 1 = (m + 1)c(S, T) + k$$

where k is the number of edges crossing the cut (S, T).

Let (S^*, T^*) be a minimal cut in G' having k^* edges.

Then for any cut (S, T) with k edges,

$$c'(S^*, T^*) \leq c'(S, T)$$
$$(m + 1)c(S^*, T^*) + k^* \leq (m + 1)c(S, T) + k$$
$$c(S^*, T^*) + \frac{k^*}{m + 1} \leq c(S, T) + \frac{k}{m + 1}$$

Here, both k^* and k are less than $m + 1$, and c values are integral values. For the last inequality to hold, there are only these two cases:

1. $c(S^*, T^*) < c(S, T)$. (Since both of them are integers, they differ by at least 1, so adding fractional values does not affect the inequality.)

2. $c(S^*, T^*) = c(S, T)$. This implies $k^* \leq k$.

So, for any cut (S, T), either (S^*, T^*) has a lower capacity, or if it has an equal capacity, it has a lower or same number of edges. This proves that a minimum cut in G' is a minimal cut in G with a minimum number of edges.

10. **[Uniqueness of Minimum Cut]** Design an algorithm to check if a flow network has a unique minimum capacity $s - t$ cut, and if not, then output two different minimum capacity $s - t$ cuts.

Run a maximum flow algorithm and compute the maximum flow f.

Let $S^* = \{v \in V \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = \{v \in V \mid t \text{ is reachable from } v \text{ in } G_f\}$. Since s and t are disconnected, $S^* \cap T^* = \phi$

Lemma 5: If (S, T) is a min-cut, there can be no edge crossing the cut in G_f.

Proof: By Max-Flow Min-Cut Theorem, $c(S, T) = |f| \Rightarrow c(S, T) = f(S, T)$. So we have

$$\sum_{\substack{u \in S \\ v \in T}} c(u, v) = \sum_{\substack{u \in S \\ v \in T}} f(u, v)$$

$$\sum_{\substack{u \in S \\ v \in T}} (c(u, v) - f(u, v)) = 0$$

Each term of this summation is nonnegative, so each term must be zero. So, $c(u, v) = f(u, v) \Rightarrow c_f(u, v) = 0$ for all pairs of nodes $(u, v), u \in S, v \in T$. So, none of these edges exist in G_f.

Converse of Lemma 5: Let (S, T) be a cut, and there are no edges crossing the cut in G_f. Then, (S, T) is a min-cut.

Proof: For all edges crossing the cut, $c_f(u, v) = 0, \Rightarrow f(u, v) = c(u, v)$. So, $c(S, T) = f(S, T) = |f|$, thus (S, T) is a min-cut.

Theorem: If $S^* \cup T^* = V$, then G has a unique min-cut.

Proof: Clearly $S = S^*, T = T^*$, (S, T) is a min-cut of G. $c(S, T) = |f|$. Assume that there exists another min cut (S', T'). WLOG, assume there exists some $u \in S', u \notin S$. Since $u \notin S, u \in T$, so t is reachable from u in G_f. Consider the path from u to t in G_f, let it be $\langle u = v_0, v_1, v_2, ..., v_{k-1}, v_k = t \rangle$. Let v_i be the first node in this path which belongs to T'. Then, $v_{i-1} \in S'$. The edge (v_{i-1}, v_i) crosses the (S', T') cut. This is a contradiction (by Lemma 5). Thus, there is a unique min-cut.

Compute S^* and T^* (by traversing the residual graph from s and the reverse residual graph from t). If their union is V, then there exists a unique cut. Otherwise the following are valid different $s - t$ cuts (by Converse of Lemma 5):

- $(S^*, V - S^*)$
- $(V - T^*, T^*)$

11. **[Edge Capacity Increase]** We are given a flow network and a maximum flow in that network. Assume all edge capacities are positive integers. Suppose we increase the capacity of an edge by 1. Give an $O(m + n)$ time algorithm to update the maximum flow.

Let G be the original network, and G' be the network after increasing an edge's capacity by 1. Let f be a max-flow in G.

Compute the residual graph G'_f. Find an augmenting path in G'_f. There will be at most one augmenting path, with residual capacity 1. If a path exists, augment along this path, the resulting flow is a max-flow of G'. Otherwise, f is already a max-flow of G'.

This runs in $O(n+m)$ time.

Proof: In G_f, s and t are disconnected. When we construct G', we increase $c(u,v)$ by 1 for some (u,v). This means in G'_f, we have $c'_f(u,v) = c_f(u,v) + 1$ for that (u,v).

We have two cases:
1. No $s-t$ path exists in G'_f. Then clearly, f is a max-flow of G'.
2. An $s-t$ path exists in G'_f. This path must use the (u,v) edge. Also, this means $c_f(u,v) = 0$ (otherwise f won't be a max flow of G) and $c'_f(u,v) = 1$.

We augment along this path to get a flow with value $|f'| = |f| + 1$. We also need to show no further augmentations are possible.

Let (S,T) be a min-cut in G, $c(S,T) = |f|$. In G', capacity of an edge was increased by 1. So, $c'(S,T) \leq c(S,T) + 1 = |f| + 1$. By Lemma 2, $|f'| \leq c'(S,T) \leq |f| + 1$. Thus, we cannot increase $|f'|$ any more, and the current flow is the max-flow of G'.

12. **[Submodularity of Cuts]** Let $G(V,E)$ be an undirected and unweighted graph. For $X \subseteq V$, we define $\delta(X)$ to be the capacity of the cut $(X, V \setminus X)$. Then show the following for every two subsets $A, B \subseteq V$,

$$\delta(A) + \delta(B) \geq \delta(A \cup B) + \delta(A \cap B)$$

We take each edge to have a capacity of 1. Since the graph is unweighted we have $c(A,B) = c(B,A)$. All other properties of c (as proved above) hold.

$$\delta(X) = c(X, V \setminus X)$$

Let $P_1 = A - B$, $P_2 = A \cap B$, $P_3 = B - A$ and $P_4 = V - A - B$. P_is are mutually disjoint and partition the set of vertices V.

$$\begin{aligned}
\delta(A) = c(A, V-A) &= c(P_1 \cup P_2, P_3 \cup P_4) = c(P_1,P_3) + c(P_2,P_3) + c(P_1,P_4) + c(P_2,P_4) \\
\delta(B) = c(B, V-B) &= c(P_2 \cup P_3, P_1 \cup P_4) = c(P_2,P_1) + c(P_3,P_1) + c(P_2,P_4) + c(P_3,P_4) \\
\delta(A \cup B) = c(A \cup B, V - A \cup B) &= c(P_1 \cup P_2 \cup P_3, P_4) = c(P_1,P_4) + c(P_2,P_4) + c(P_3,P_4) \\
\delta(A \cap B) = c(A \cap B, V - A \cap B) &= c(P_2, P_1 \cup P_3 \cup P_4) = c(P_2,P_1) + c(P_2,P_3) + c(P_2,P_4)
\end{aligned}$$

So,

$$\begin{aligned}
\delta(A) + \delta(B) &= c(P_1,P_2) + 2c(P_1,P_3) + c(P_1,P_4) + c(P_2,P_3) + 2c(P_2,P_4) + c(P_3,P_4) \\
&\geq c(P_1,P_2) + c(P_1,P_4) + c(P_2,P_3) + 2c(P_2,P_4) + c(P_3,P_4) \\
&= \delta(A \cup B) + \delta(A \cap B)
\end{aligned}$$

13. **[Edge in Every Min Cut]** Given a flow network, design an algorithm of worst-case time complexity $O(\text{time complexity of maximum } s-t \text{ flow computation})$ that identifies all edges that belong to **every** minimum $s-t$ cut.

Run a maximum flow algorithm and compute the maximum flow f.

Let $S^* = \{v \in V \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = \{v \in V \mid t \text{ is reachable from } v \text{ in } G_f\}$.

Compute S^* and T^* by running BFS on the residual graph and its reverse graph respectively. Iterate over all edges, and check whether $u \in S^*$ and $v \in T^*$, if so, then (u,v) belongs to every min-cut of G.

This algorithm runs in $O(\text{Max-Flow})$ time.

Lemma 6: If $(u, v) \in E$ and $u \in S^*$, $v \in T^*$, then (u, v) crosses all min-cuts of G.

Proof: Let $(u, v) \in E, u \in S^*, v \in T^*$. Let (S, T) be a min-cut of G, such that (u, v) does not cross the cut. WLOG, let $u, v \in S$. Since $v \in T^*$, v is connected to t in G_f. Let $\langle v_0 = v, v_1, v_2, ..., v_k = t \rangle$ be the path from v to t in G_f. Let v_i be the first vertex which is in T. Then, (v_{i-1}, v_i) crosses the cut (S, T). By Lemma 5, (S, T) can't be a min-cut, which is a contradiction. Thus, (u, v) crosses every min-cut of G.

Converse of Lemma 6: If $(u, v) \in E$ crosses all min-cuts of G, then $u \in S^*$ and $v \in T^*$.

Proof: Since (u, v) crosses all min-cuts, it must cross $(S^*, V - S^*)$, which means $u \in S^*$. It must also cross $(V - T^*, T^*)$, which means $v \in T^*$.

14. **[Flow with a Mandatory Edge]** Given a network and a particular edge $e = (u, v)$, design an algorithm that determines whether there exists a maximum $s-t$ flow that sends strictly positive flow through e.

1. Run a standard algorithm to compute any maximum $s - t$ flow f in the network G.
2. If $f(u, v) > 0$, then we have already found a maximum flow that sends positive flow through e. Return true.
3. If $f(u, v) = 0$, construct the residual graph G_f.
4. Note that since $f(u, v) = 0$, the forward edge (u, v) has its full residual capacity $c_f(u, v) = c(u, v) > 0$, meaning the directed edge (u, v) exists in G_f.
5. Run a DFS in G_f starting from v to check if u is reachable from v.
6. If u is reachable from v, there exists a path from v to u, which means there is a cycle in G_f containing (u, v). Pushing a small amount of flow $\delta > 0$ along this cycle strictly increases the flow on (u, v) without changing the total $s - t$ max flow value (since it is a closed circulation) and without violating any capacity constraints. Thus, return true.
7. If u is not reachable from v in G_f, no such cycle exists. This means there is no way to reroute the existing flow to use (u, v) without decreasing the overall $s - t$ flow. Return false.

Time Complexity: This algorithm takes $O(\text{Max-Flow})$ time to find f, and $O(V + E)$ time for the DFS on the residual graph. The overall time complexity is strictly bounded by $O(\text{Max-Flow})$.

15. **[Path decomposition of a flow]** A flow f in a flow network G is called *not acyclic* if there exists a directed cycle in G such that each of its edges carries a positive amount of flow. Otherwise, f is called *acyclic*.
    (a) Prove that, for every flow f, there exists an acyclic flow f' of value the same as the value of f.
    (b) A path flow is a flow which assigns a positive flow only to the edges of an $s - t$ path. Prove that, every acyclic flow f can be written as the sum of at most m path flows.
    (c) Is the Ford-Fulkerson algorithm guaranteed to find an acyclic flow?
    (d) A cycle flow is a flow which assigns a positive flow only to the edges of a directed cycle. Prove that, every flow f can be written as the sum of at most m path flows and cycle flows. Given a flow network and a flow f, design an algorithm to find the above decomposition in $O(mn)$ time.

(a) Let f be a flow in G. We define a new flow $f' = f$. Define the flow subgraph G^+ containing only edges with positive flow, $f'(u, v) > 0$.

While G^+ has a directed cycle C,
- Let $\delta = \min_{e \in C} f'(e)$.
- For every edge $e \in C$, update $f'(e) \leftarrow f'(e) - \delta$.
- The flow of an edge becomes zero, which gets removed from G^+.

This procedure will terminate in at most m iterations since on each iteration one edge gets removed from the graph G^+.

f' is a valid flow since for each node in the cycle, one incoming edge and one outgoing edge is depleted of δ flow on each iteration.

The resulting flow f' is acyclic flow, since the flow graph G^+ does not contain any directed cycle at the end of the algorithm.

The value of the flow is same as that of the original, since we do not modify the flow of any edge connected to s – under our assumptions, s can't be a part of a directed cycle.

(b) Let f be an acyclic flow in G. Consider the following algorithm to extract path flows from f.

Let $f' = f$. While there is a $s - t$ path in G^+,
- Let $\delta = \min_{e \in p} f'(e)$.
- For every edge $e \in p$, update $f'(e) \leftarrow f'(e) - \delta$.
- Add the path flow p with flow value δ to the list of path flows.
- The flow of an edge becomes zero, which gets removed from G^+.

This procedure will terminate in at most m iterations since on each iteration one edge gets removed from the graph G^+.

Also, at the end of the procedure, (when there is no $s - t$ path in G^+), f' must be the zero flow.

Assume it is not, so there exists some $(u, v) \in E, f(u, v) > 0$. Since v has an incoming positive flow, it must have an outgoing positive flow through some edge (v, v_1). Repeating this argument, we can find a sequence $v, v_1, v_2, \ldots$. Now, no vertex can repeat in this sequence, otherwise we will have a directed cycle with positive flow, which is not allowed. So the only other option is that this sequence terminates at t.

Similarly, traceback the vertex u to $u_1, u_2, u_3, \ldots$, this leads to this sequence terminating at s. Therefore, we have found an $s - t$ path: $s \rightsquigarrow u \to v \rightsquigarrow t$, which is a contradiction. So, f' is indeed the zero flow.

Thus we have decomposed f into at most m path flows.

(c) No. We show a counter-example:

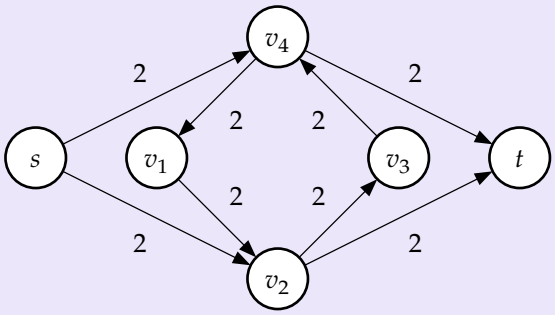

Figure 10: A network for which FORD-FULKERSON may not find a acyclic flow.

On this network, if we choose the path $\langle s, v_2, v_3, v_4, t \rangle$ on the first iteration, and $\langle s, v_4, v_1, v_2, t \rangle$ on the second iteration, we will end up with a flow which is not acyclic, since there is a directed cycle with positive flow, v_1, v_2, v_3, v_4.

(d) The proof follows directly from (a) and (b).

Define $f' \leftarrow f$. Let G^+ contain edges where $f'(u,v) > 0$.

While $f' \neq 0$,
- Perform DFS from s on G^+.
  - If a back edge is encountered, we have a directed cycle.
    - Decrease the flow of each edge in the cycle by δ, the bottleneck edge.
    - Store this as a cycle flow.
    - Restart the DFS.
  - If t is reached, we have a $s-t$ path.
    - Decrease the flow of each edge in the path by δ, the bottleneck edge.
    - Store this as a path flow.
    - Restart the DFS.

This runs in $O(mn)$ time, since on each iteration at least one edge gets removed, so there are at most m iterations. Each iteration runs in $O(n)$ since we stop once we either reach t, or a back edge. So we never backtrack on the DFS, we stop after visiting at most n vertices.

16. **[Dinic's Algorithm]** A *blocking flow* f in a flow network $G(V, E, c : E \to \mathbb{R}_{>0})$ is a flow such that, in every $s-t$ path, there is at least one edge e such that $c_e = f_e$.
    (a) Given a network G, and an $s-t$ flow f, construct the BFS tree starting from s. Delete backward and cross edges of the BFS tree from the residual graph G_f. We call the resulting graph the layered graph $\mathcal{L}_f$ with respect to f. Prove that $\mathcal{L}_f$ is acyclic.
    (b) Design an $O(mn)$-time algorithm to compute a blocking flow f in $\mathcal{L}_f$. *Hint: Modify the standard DFS appropriately!*
    (c) Consider the following algorithm for computing a maximum flow in a network. Start with the zero flow. Compute a blocking flow in the corresponding layered graph, augment it with the current flow, and iterate until there is no path from s to t in the residual graph. Prove that the run time of this algorithm $O(mn^2)$. *Hint: How does the distance (number of edges in a shortest path) of any vertex from s changes after every iteration?*

(a) Define $\ell(v) = \delta_{G_f}(s,v)$ = length of shortest path from s to v in G_f. Back edges and cross edges in the BFS tree are those (u,v) which have $\ell(v) \leq \ell(u)$. If we remove these edges in G_f, the only edges we keep are (u,v) such that $\ell(v) = \ell(u) + 1$.

Suppose there is a directed cycle in $\mathcal{L}_f$: $v_1, v_2, ..., v_k = v_0$, then since each edge increases ℓ by 1, $\ell(v_1) < \ell(v_2) < ... < \ell(v_k) = \ell(v_0)$, which is a contradiction.

Thus, $\mathcal{L}(f)$ is acyclic.

(b) To compute the blocking flow in $O(mn)$ time, we use a modified Depth First Search (DFS) with a "current-edge pointer" for each vertex.

Algorithm:
1. For each vertex $u \in V$, initialize a pointer `ptr[u]` pointing to the first outgoing edge in its adjacency list in $\mathcal{L}_f$.
2. Start a DFS from s. At the current vertex u, look at the edge $e = (u,v)$ pointed to by `ptr[u]`.
3. If $c_f(e) > 0$, we advance our DFS to v.
4. If we reach t, we have found an $s-t$ path in $\mathcal{L}_f$.
   - Find the bottleneck capacity δ of this path.
   - Augment the flow along this path by δ, updating the residual capacities. (This saturates at least one edge).
   - Restart the DFS from s.

5. Let $e = (u, v)$. If $c_f(e) = 0$, or if v is a dead end (meaning `ptr[v]` has reached the end of its adjacency list or it has no outgoing edges), then increment `ptr[u]` and restart the DFS from s.
6. The algorithm terminates when `ptr[s]` reaches the end of s's adjacency list.

Complexity: Because we never reset `ptr[u]`, an edge is removed permanently once it is saturated or leads to a dead end. Every DFS starts from s and takes at most n steps. Each DFS run can end in these ways:

- It reaches t. We augment flow, which removes at least one edge from $\mathcal{L}_f$. Since there are at most m edges, this happens at most m times. Time spent on this case across the whole algorithm is bounded by $m \times O(n) = O(mn)$.
- It hits a dead end (or a saturated edge). We advance `ptr[u]` and discard that edge from future consideration. Since there are at most m edges, `ptr` arrays can be advanced at most m times total. Time spent on this case is bounded by $m \times O(n) = O(mn)$.

Thus, the total time complexity is exactly $O(mn)$.

(c) To prove the $O(mn^2)$ runtime, we must bound the total number of blocking flow iterations.

Let $\ell(v)$ be the shortest path distance (number of edges) from s to v in G_f. By definition, every edge (u, v) in $\mathcal{L}_f$ strictly satisfies $\ell(v) = \ell(u) + 1$.

When we compute and augment a blocking flow in $\mathcal{L}_f$, we saturate at least one edge on **every** possible shortest $s - t$ path. Therefore, in the new residual graph $G_{f'}$, no $s - t$ paths exist that are made entirely of the forward edges from the previous $\mathcal{L}_f$.

Any newly created edges in $G_{f'}$ are backward edges (v, u), which point strictly backwards relative to the distance layering: $\ell(u) = \ell(v) - 1$.

For a new $s - t$ path to exist in $G_{f'}$, it must use either these backward edges or cross edges (where $\ell(v) \leq \ell(u)$). Because traversing these edges does not optimally increase the distance from s by 1 at each step, any valid $s - t$ path in $G_{f'}$ must require strictly more edges than the shortest path in G_f.

Thus, after every iteration of augmenting a blocking flow, the shortest path distance $d(t)$ strictly increases.

$$d_{f'}(t) > d_f(t)$$

Since the maximum possible length of a simple path in a graph with n vertices is $n - 1$, the distance $d(t)$ can strictly increase at most $n - 1$ times before s and t become completely disconnected.

This guarantees there are at most $O(n)$ blocking flow iterations. Since each iteration computes a blocking flow in $O(mn)$ time (from part b), the total time complexity of Dinic's Algorithm is:

$$O(n) \text{ iterations} \times O(mn) \text{ time per iteration} = O(mn^2)$$