Algorithms Fall 2019

Lecture : Network Flows

Imdad Ullah Khan *Scribe: Imdad Ullah Khan*

Contents

1 Motivation and Definition

Consider a network of pipelines in Figure 1 along which oil can be sent from the source s to the sink t. The goal is to ship as much oil from s to t as possible. There are two restrictions: a given pipeline cannot carry more oil than the weight of the corresponding edge. Secondly, no node can store any oil.

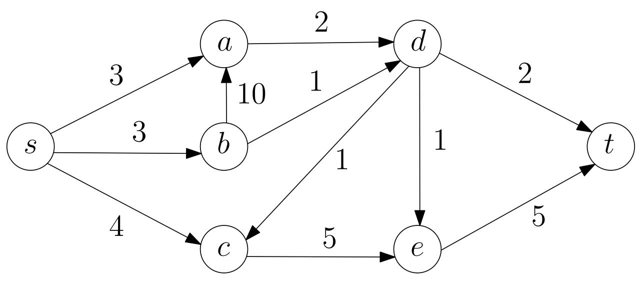


Figure 1: An example network

In this network, we can send 2 units of flows along the path s, a, d, t. Another 2 units of flow can go through the path s, c, e, t, as shown in Figure 2

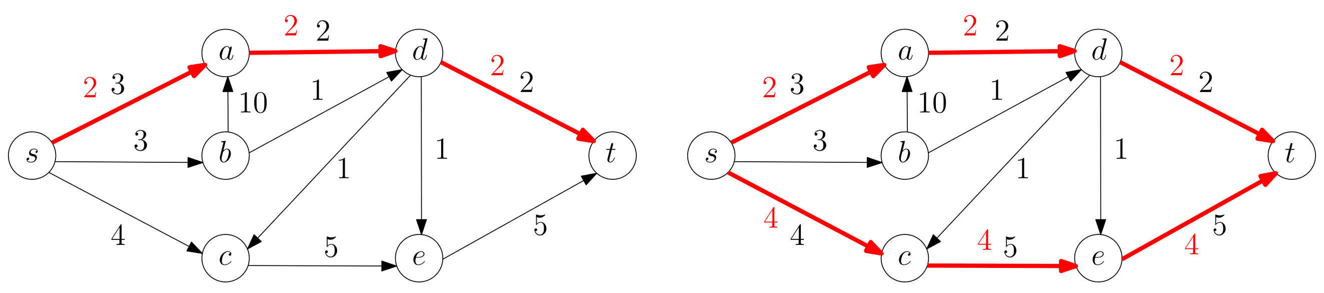


Figure 2: Flow in a network

It is natural to ask the following about the flow in the network. Is this the best flow, i.e. is this the largest amount of oil that can be shipped? How do we measure the size of flow? How do we determine that a given flow is the maximum possible? In order to answer such questions, we first formalize the problem.

1.1 Formal definition of max flow problem

A flow network is a directed graph $G = (V, E)$. Such a graph typically model a transportation network whose edges carry traffic and vertices serve as switches to pass traffic between different edges. Examples could be highway networks, where edges are roads and vertices are interchanges or intersection, a communication networks where edges are links and vertices are switches, a fluid networks where edges are pipelines and vertices are junctures where pipes are plugged together. Such networks typically have capacities on edges indicating how much traffic can they carry, source nodes where traffic is generated, and target nodes where traffic is consumed. We formalized all of these below.

Each edge $e = uv \in E$ is associated with a nonnegative real number $c_e = c_{uv}$ called the capacity of the edge uv. Two designated nodes s (traffic generator) and t (traffic consumer), where s is a source ($deg^-(s) = 0$) and t is a sink ($deg^+(t) = 0$).

A $s-t$ flow or traffic through the network is given by assigning each edge e of G a real number f_e which does not exceed c_e. Formally, a flow f is a mapping from E to real numbers $f : E \to \mathbb{R}$ satisfying the capacity and storage constraints. We denote by f_e the value of f at e, i.e. $f_e = f(e)$. The two constraints on the flow are as follows:

- capacity constraints: $\forall\ e \in E \quad : \quad 0 \leq f_e \leq c_e$
- flow conservation constraints: $\forall\ v \in V\ , v \neq s, t$

$$\underbrace{\sum_{e \text{ into } v} f_e}_{\text{total flow incoming to } v} \quad = \quad \underbrace{\sum_{e \text{ out of } v} f_e}_{\text{total flow outgoing from } v}$$

Basically, the capacity constraints says that flow through each edge is non-negative and cannot exceed its capacity. The flow conservation constraints says that excepts for s and t the total flow entering a vertex v, which is the sum of flow over all edges incoming to v is equal to the total flow outgoing form v which is the sum of flow over all outgoing edges from v.

The value or size of a flow f is the total amount of flow generated or sent out from s, i.e.

$$size(f) = \sum_{e \text{ outgoing from } s} f_e$$

For notational convenience for a vertex v we define

$$f^{out}(v) = \sum_{e \text{ outgoing from } v} f_e \qquad \text{and} \qquad f^{in}(v) = \sum_{e \text{ incoming to } v} f_e.$$

Furthermore, for a set $X \subset V$, we define

$$f^{out}(X) = \sum_{e \text{ outgoing from } X} f_e \qquad \text{and} \qquad f^{in}(X) = \sum_{e \text{ incoming to } X} f_e.$$

With this notation

$$size(f) = f^{out}(s).$$

Using flow conservation constraints, that we get

$$size(f) = f^{out}(s) = f^{in}(t).$$

In the example in Figure 2 the size of the flow is 6.

Problem 1 (Maximum-Flow)**.** *Given a flow network, find f such that $size(f)$ is as large as possible.*

1.2 Structural bounds on size of a flow

Consider our running example network from Figure 1 and consider the cut $[\{s\}, \overline{\{s\}}]$ as shown in Figure 3. Any flow that is generated from s, has to go through one of these three edges, hence no flow can be of size bigger than $3 + 3 + 4 = 10$.

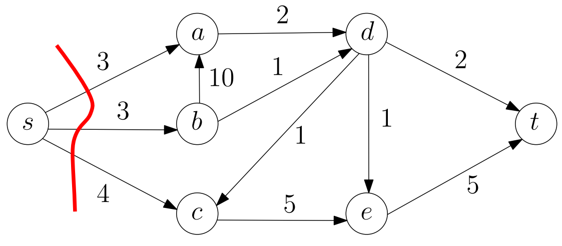


Figure 3: cut $[\{s\}, \overline{\{s\}}]$

The same is true for any cut, consider the cut $[\{s, a, b, c\}, \{d, e, t\}]$ in Figure 4. Since s is in the left side and t is in the right, any $s - t$ flow must cross this cut (hence use one of the edges from left side to the right), there cannot be a flow in this network of size more than $2 + 1 + 5 = 8$. This is a tighter bound than the one we got from the other cut.

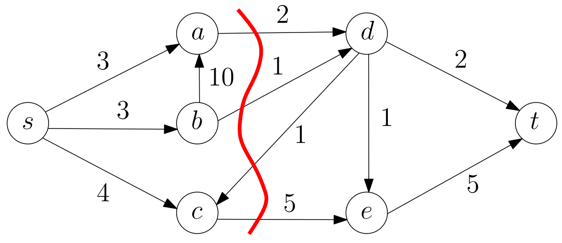


Figure 4: cut $[\{s, a, b, c\}, \{d, e, t\}]$

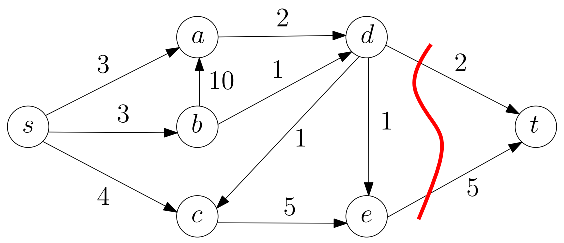


Figure 5: cut $[\overline{\{t\}}, \{t\}]$

Consider yet another cut $[\overline{\{t\}}, \{t\}]$ as in Figure 5. By the same reasoning as above we get that any flow is of size at most 7.

Notice that in all these cuts we have s on one side and t on the other side for obvious reason, these are called $s-t$ cuts. We formally define it as follows. A $s-t$ cut, denoted by $[A, \overline{A}]$ is a cut in the graph, i.e. $A \subset V$ with the restriction that $s \in A$ and $t \in \overline{A}$. Capacity of a $s-t$ cut is the sum of capacities of edges going from A to $\bar{A}$, i.e.

$$c([A, \overline{A}]) = \sum_{e \text{ outgoing from } A} c_e.$$

From the above examples, we get the following upper bound on any flow in G.

Lemma 2. *Let f be a flow in G and let $[A, \bar{A}]$ be any $s-t$ cut in G, then*

$$size(f) \leq c([A, \overline{A}]).$$

Proof. Let $[A, \overline{A}]$ be any cut. By definition we know that

$$\begin{aligned}
size(f) &= f^{out}(s) \\
&= f^{out}(s) + \sum_{s \neq v \in A} \left(f^{out}(v) - f^{in}(v)\right) && \text{just adding a few 0's} \\
&= f^{out}(s) + \sum_{s \neq v \in A} f^{out}(v) - \sum_{s \neq v \in A} f^{in}(v) \\
&= f^{out}(A) - f^{in}(A) \\
&= \sum_{e \text{ outgoing from } A} f_e - \sum_{e \text{ incoming to } A} f_e && \text{flows on all other edges cancel} \\
&\leq \sum_{e \text{ outgoing from } A} c_e - \sum_{e \text{ incoming to } A} f_e \\
&\leq \sum_{e \text{ outgoing from } A} c_e \\
&= c([A, \overline{A}]).
\end{aligned}$$

□

It is also clear that although any cut provide an upper bound on the size of a flow, since we are interested in the most tight bound, it makes sense to consider a $s-t$ cut of minimum capacity. In other words let $[A^*, \overline{A^*}]$ be a $s-t$ cut with minimum capacity, (this is called a Min-Cut). We get that

Fact 3.

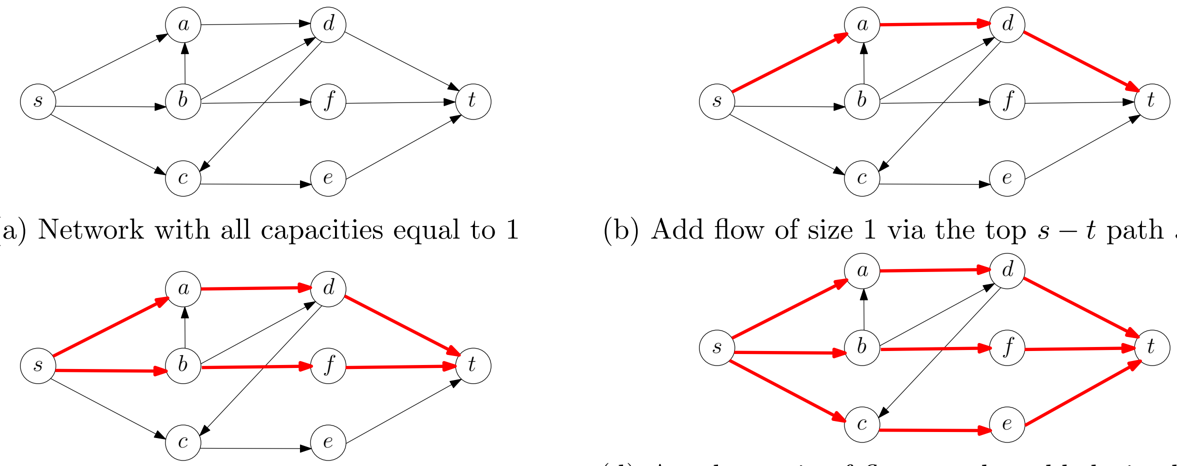
$$size(f) \leq c([A^*, \overline{A^*}]).$$

2 Designing an Algorithm

There is no known dynamic programming algorithm for max-flow. We try out a greedy strategy.

2.1 A Greedy Approach

The greedy approach builds up flow little bit at a time. Starting with a 0 flow, we add flow via a path. We observe that it satisfies both the capacity and flow conservation constraints. Consider the network and flow addition in Figure 6.



(a) Network with all capacities equal to 1

(b) Add flow of size 1 via the top $s-t$ path s, a, d, t

(c) Flow can be added via the path s, b, f, t.

(d) Another unit of flow can be added via the path s, c, e, t

Figure 6: Adding flows via paths

Can more flow be added? Notice that there is a $s-t$ cut in this network of capacity 3 as shown in Figure 7, hence this flow is the maximum possible.

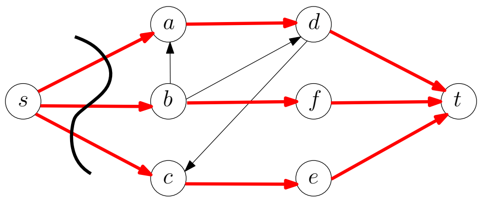


Figure 7: $s-t$ cut of capacity 3

The capacities don't have to be 1 for such a greedy algorithm to work. Consider the network in Figure 8 with all capacities 1 except the edge dt with capacity 2. Consider the path s, a, d, t. The bottleneck (smallest capacity of an edge) in this path is 1, so we add a flow of size 1 via this path. Next we add a flow of size 1 via the path s, c, e, t. Next we add one unit of flow via the path s, b, d, t. reusing the edge dt still not violating the capacity constraint.

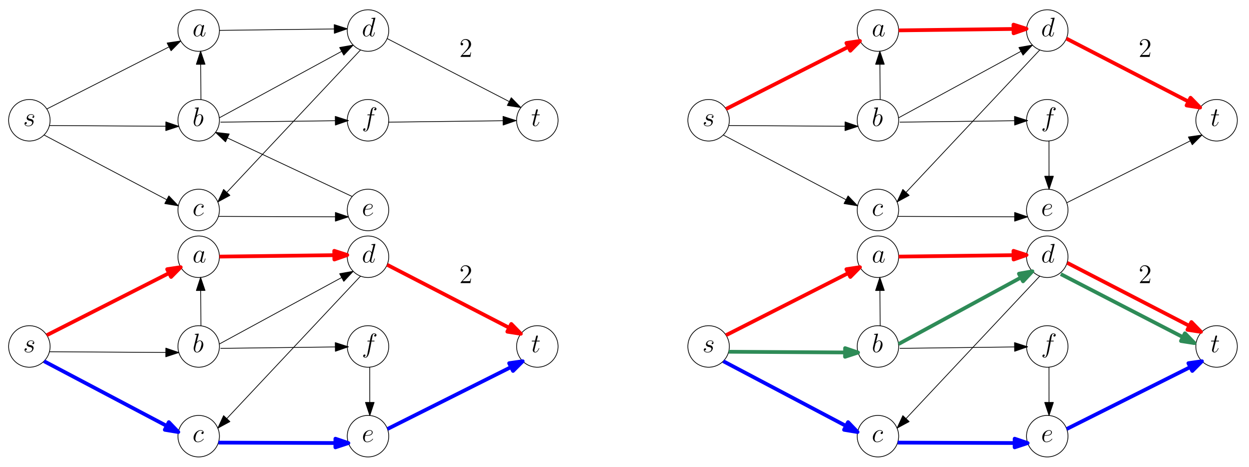


Figure 8: Greedy algorithm works for capacities greater than 1

The two shown cuts shown in Figure 9 that this indeed is a max-flow, because the min-cut in this graph is of size 3.

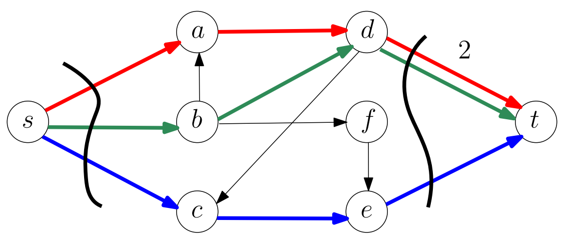


Figure 9: Min-Cut is of size 3

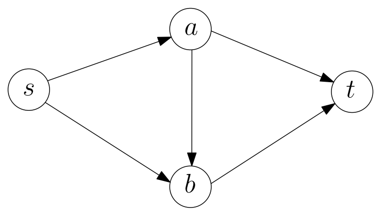
The important things to note here are:

- **Adding a flow via a path equal to the bottleneck of that path doesn't violate the capacity constraints.**
- **Adding flow via paths ensures that flow conservation constraints remain satisfied**
- **Both of the above facts ensures that any intermediate step of such a greedy algorithm the flow indeed is a valid flow**

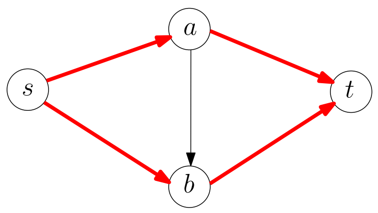
2.2 Problem with Greedy Approach

Next, we identify problem with this algorithm, i.e. show with an example that this approach doesn't necessarily construct a max-flow. Then, we give a fix for the problem and design and analyze the final algorithm.

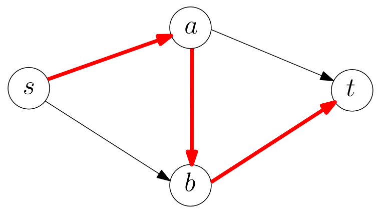
Consider the example in Figure 10. The given network has only 4 vertices, 5 edges and all the capacities equal to 1. The max flow clearly is of size 2. But if our greedy algorithm first chose the path s, a, b, t which is a valid $s - t$ path we do not have any more path, remaining in the graph to send flow over. At least not a path with non-zero remaining capacity. Note that the problem is not the length of the path i.e. the number of hops in the path, as in the following graph our algorithm could still get stuck. Here we cannot directly add another unit of flow because there is no $s - t$ path on which we can send some flow yet do not exceed some capacity. Therefore, we need something more.



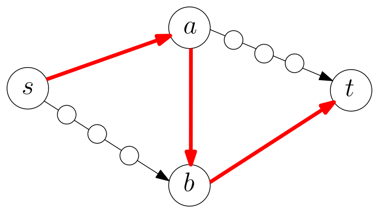
(a) Given network, all capacities 1



(b) Max-flow is of size 2



(c) Max-flow not produced if path s, a, b, t is chosen

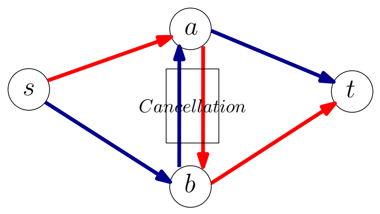


(d) Problem is not the length of a path

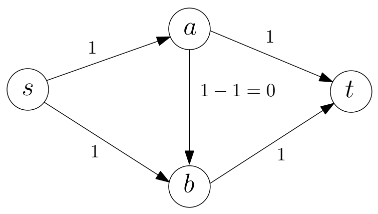
Figure 10: Instance where the greedy algorithm so far does not produce max-flow

A more general way of pushing further flow is that we can push forward flow on edges where some capacity is remaining. At the same time we will cancel flow on the edges already carrying some flow, or think of it as pushing flow backward.

Consider the same example as in Figure 10 and suppose one unit of flow is already added on the path s, a, b, t. If we think of canceling the flow on the edge ab, then we can add another unit. Figure 11 illustrates this technique. We can add another unit of flow via the path s, b, a, t (the blue path). But the edge ba doesn't exist in the network, so we cannot send any flow from b to a, but we can cancel the already existing flow on the edge a to b. In this sense we end up getting the maximum flow in this network, where all edges have flow 1, except the edge ab, where the red flow of 1 unit is canceled by the blue flow of 1 unit. Adding the red and blue flow, we get a total flow of size 2. Note that this is the maximum flow and doesn't violate any constraint.



(a) Blue flow cancels the red flow



(b) Total flow is the max-flow

Figure 11: Flow cancellation is the fix to the problem identified in Figure 10

It turns out that this cancellation of already existing flows over certain edges indeed is the right framework to add more flow to an existing flow. A systematic way to search for adding and canceling flow is to use the so-called residual network.

2.3 Residual Network

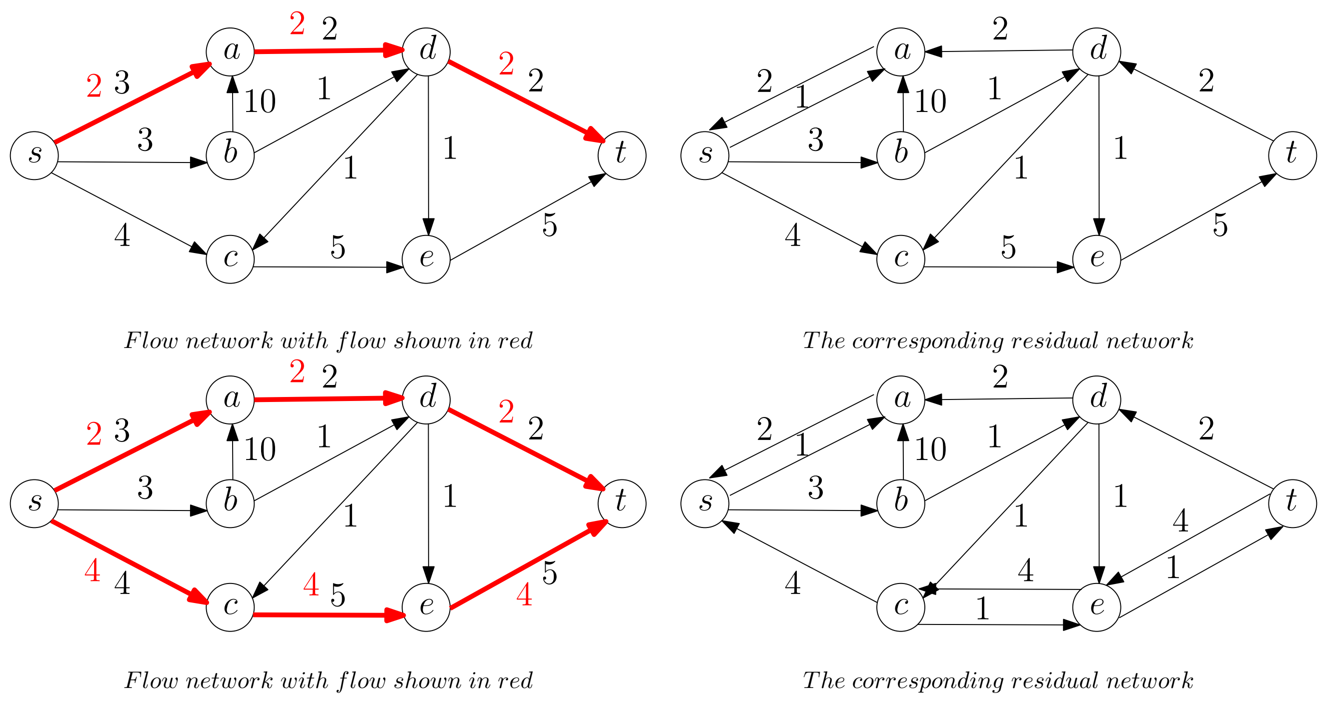
A residual network associated with a given flow, is a new network that represent places where more flow can be added. This not only include edges where there is remaining room to add flow but also edges where existing flow on some edges can be canceled.

Given a network G and a flow f on G, the residual graph G_f of G with respect to f is defined as follows:

- Vertex set of G_f is the same as that of G
- For each $e = uv$ of G on which $f_e < c_e$, there is an edge $e = uv$ in G_f with a capacity $c_e - f_e > 0$. These are called forward edges since on these edges we can push forward $c_e - f_e$ units of flow. $c_e - f_e$ is the leftover or residual capacity on e.
- For each edge $e = uv$ of G on which $f_e > 0$, there is an edge $e' = vu$ in G_f with a capacity of f_e. These are called backward edges (note that the direction is reversed). $c_{e'}$ is f_e since on e we can cancel or push backward f_e units of flow.

Note that each edge $e \in E(G)$ gives rise to one or two edges depending on f_e and c_e. So G_f has at most twice as many edges as G. Since a zero flow ($f_e = 0$ for all edges) is a valid flow, the residual graph w.r.t the zero flow by the above definition is the same as G itself.

In Figure 12, we give a couple of examples of residual networks.



Flow network with flow shown in red *The corresponding residual network*

Flow network with flow shown in red *The corresponding residual network*

Figure 12: Example flows and corresponding residual networks

2.4 Augmenting Paths

A simple $s - t$ path (no vertex repeated) in the residual graph is called an augmenting path, (because it is used to augment an existing flow). For a flow f and an augmenting path P in G_f, let $bottleneck(P, f) = \min_{e \in P} c'_e$ be the minimum residual capacity of any edge on P in G_f. Here c'_e is the residual capacity of the edge e (its capacity in G_f). We use the following augment procedure to add flow to f.

Algorithm 1 : Augment the flow f using the path P to output a flow f'

```
function AUGMENT($P, f$)
    $b \leftarrow bottleneck(P, f)$
    $f' \leftarrow f$                                         ▷ Initialize $f'$ by $f$
    for each edge $e = uv \in P$ do
        if $e$ is a forward edge then        ▷ If $e = uv$ is a forward edge with $\geq b$ residual capacity
            $f'_e \leftarrow f_e + b$                          ▷ Add $b$ units of flow on edge $uv$
        else if $e$ is a backward edge then    ▷ so $vu$ is a forward edge with at least $b$ flow in $f$
            $f'_{vu} \leftarrow f_{vu} - b$                    ▷ Cancel $b$ units of flow on edge $vu$
```

2.5 Analysis of Augment Procedure

The only analysis we need here is to show correctness, i.e. the output of the Augment procedure f' indeed is a flow. Later, we will show that $size(f') > size(f)$ and call this function in an algorithm to find max-flow.

Lemma 4. *f' is a flow in G*

Proof. To show that f' is a flow, we need to show that f' satisfies capacity constraint on all edges of G and that it satisfies flow conservation constraint on all vertices of G.

Since f is a flow, capacity constraints on all edges not on P remain satisfied in f' too (as it is just a copy). We only made changes to edges on P.

Let $e = uv$ be an edge on P.

If e is a forward edge in G_f, then by construction $f'_e = f_e + b$. By definition $0 < b \leq c_e - f_e$. We have

$$0 \leq f_e \leq f'_e = f_e + b \leq f_e + c_e - f_e = c_e$$

On the other hand, if $e = uv$ is a reverse edge with residual capacity f_e, then by construction of G_f the edge vu has a flow f_e. So using the fact that $0 < b \leq f_e$ we have

$$c_e \geq f_e \geq f'_e = f_e - b \geq f_e - f_e = 0.$$

Hence in both cases we get that the capacity constraint is satisfied on the edge e.

To show that flow conservation constraints are satisfied; again note that every thing remain the same for vertices not on P. For a vertex $v \neq s, t$ on P there must be an edge of P incoming edge e_{in} to v and an outgoing edge e_{out} from v (as it is an internal vertex on the path). Since f satisfied conservation condition on v. We will show that the change in flow entering v by f' is the same as the flow exiting v.

There are four cases depending on whether e_{in} and e_{out} are forward or reverse edges. We will discuss one or two of these cases the rest are very similar and mechanical proofs using definitions.

Suppose both $e_{in} = xv$ and $e_{out} = vy$ are forward edges. Then by design $f'^{out}(v) = f^{out}(v) + b$ as we $f'_{vy} = f_{vy} + b$ and $f'^{in}(v) = f^{in}(v) + b$ as $f'_{xv} = f_{xv} + b$. Hence the change in the flow incoming to v is the same as the change in flow outgoing from v.

In the case that $e_{in} = xv$ is a reverse edge and $e_{out} = vy$ is a forward edge. Then by design $f'^{out}(v) = f^{out}(v) - b + b$ as $f'_{vx} = f_{vx} - b$ (since vx is a reverse edge) and $f'_{vy} = f_v y + b$ (since vy is a forward edge). By the fact that the path is simple (v is not repeated) we have $f'^{in}(v) = f^{in}(v) + 0$. Hence the change in the flow incoming to v is the same as the change in flow outgoing from v. □

2.6 The Ford-Fulkerson Algorithm

Now, we are ready to give a complete algorithm for finding a Max Flow. This algorithm is called the Ford Fulkerson Algorithm named after its developers.

Algorithm 2 : Ford-Fulkerson-Algorithm(G)

```
$f \leftarrow 0$                                  ▷ Initialize to a (valid) flow of size 0 (on every edge)
while TRUE do
    Compute $G_f$
    Find an $s-t$ path $P$ in $G_f$                ▷ Using e.g. BFS
    if no such path then
        return $f$
    else
        $f \leftarrow$ AUGMENT($P, f$)
```

We run this algorithm on the graph in Figure 12. The example run is shown in Figure 13 which is taken from the book, Algorithms, by Dasgupta, Papadimitriou, and Vazirani.

2.7 Analyzing the Algorithm: Termination and Running Time

So far, it is not clear whether the algorithm terminates at all. We will show that it does terminate and will analyze its running time, and later prove it's optimality.

We first show that if all capacities are integer, then all the intermediate flows are integers.

Lemma 5. *At every intermediate step of the above algorithm the flow at every edge f_e and capacities of every edge in G_f are integers.*

Proof. The proof is by induction on iteration. The statement is clearly true after iteration 0, as by construction $f_e = 0$ and $G_f = G$ where are all input capacities are integers by assumption. At a given iteration given an integral flow f and integral capacities in G_f, the capacity b of the bottleneck edge is an integer. The resulting flow hence is an integer plus/minus b hence remains integral. Similarly capacities in the residual graph. □

Next, we show that every new flow is strictly larger than the previous one.

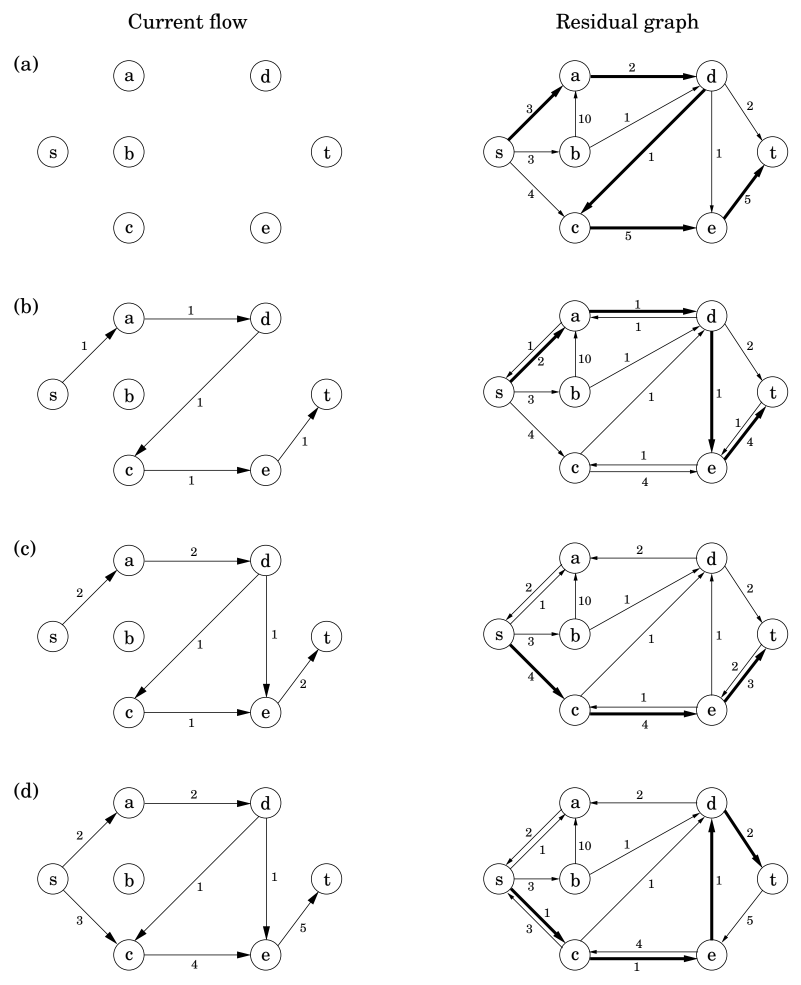
Lemma 6. *Let f be a flow in G and let P be a $s-t$ path in G_f. If f' is the flow returned by the* AUGMENT(P, f) *function, then $size(f') = size(f) + b$, where $b = bottleneck(P, f)$.*

Proof. Since $b > 0$, we immediately get that $size(f') > size(f)$. The statement of lemma follows from the definition of $size(f')$. We know that $size(f) = f^{out}(s)$. Since P is a $s-t$ path in G_f the first edge e on P is an outgoing edge from s in G_f. Furthermore, we know that $deg^-(s) = 0$ in G, hence the edge on P incident to s in G_f is a forward edge (because if sx is a backward edge in G_f, that would mean that the edge $xs \in E$ (with some flow) contradictin gthat s is a source). Hence the Augment procedure will make $f'(e) = f(e) + b$, and $size(f') = f'^{out}(s) = f^{out}(s) + b = size(f) + b$. □

Next, we show that the algorithm always terminate. By the previous lemma we know that in each iteration the flow increases, all we need to show is that the maximum flow is bounded.

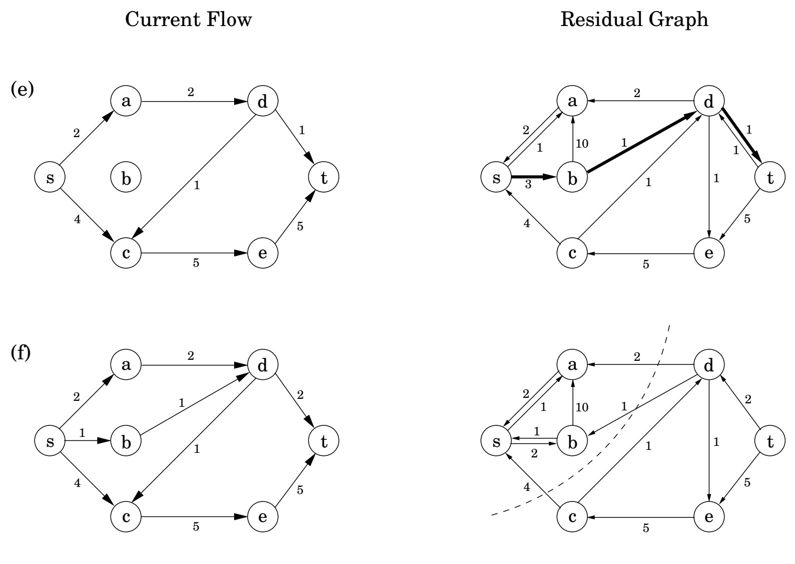
Lemma 7. *The Ford-Fulkerson Algorithm terminates in at most $C_s = c([\{s\}, \overline{\{s\}}])$*

Figure 7.6 The max-flow algorithm applied to the network of Figure 7.4. At each iteration, the current flow is shown on the left and the residual network on the right. The paths chosen are shown in bold.



[h!]

Figure 7.6 *Continued*



[h!]

Figure 13: Example Run of Ford-Fulkerson Algorithm on the network in Figure 12

Proof. By Lemma 2 any flow in G is of size at most C_s. We showed in the previous two lemmas that in each iteration the flow always increases by least an integer $b \geq 1$, hence there can be at most C_s iterations. □

Finally, we prove the runtime of the algorithm.

Lemma 8. *The Ford-Fulkerson Algorithm can be implemented in $O(mC_s)$ time.*

Proof. Since any G_f can have at most $2m$ edges we can find a $s-t$ path in a G_f in time at most $O(n+m) = O(m)$ time using a BFS or DFS from s. Besides the constant time operations AUGMENT(G, f) takes time $O(n) = O(m)$ by visiting each edge of P and making an increment or decrement operation. We showed that there are at most C_s iterations, hence this implementation takes $O(mC_s)$ time. Here we made an assumption that every vertex except s in G has at least one incoming edge, hence $m \geq n/2$. □

2.8 Optimality of Ford-Fulkerson Algorithm: The Max-Flow-Min-Cut Theorem

We saw in the proof of Lemma 2 that for any flow f and any $s-t$ cut $[A, \overline{A}]$ we have

$$size(f) = f^{out}(A) - f^{in}(A) \tag{1}$$

Furthermore, since $[A, \overline{A}]$ is a cut we know that edges into A are precisely the edges out of $\overline{A}$ and edges outgoing from A are precisely the edges incoming to $\overline{A}$ and similarly flow into A is precisely flow out of $\overline{A}$. Hence we can equivalently state (1) as: for any flow f and any $s-t$ cut $[A, \overline{A}]$ we have

$$size(f) = f^{in}(\overline{A}) - f^{out}(\overline{A}) \tag{2}$$

Aas a corollary from (1) and (2), the assumption that s is a source ($deg^-(s) = 0$) and t is a sink ($deg^+(t) = 0$, and considering the cut $[\{s\}, V \setminus \{s\}]$ and the cut $[V \setminus \{t\}, \{t\}]$ we get that

$$size(f) = f^{out}(s) = f^{in}(t) \tag{3}$$

Theorem 9. *If f is a flow such that there is no $s-t$ path in G_f, then there is a $s-t$ cut in $[A^*, \overline{A^*}]$ in G such that $size(f) = c([A^*, \overline{A^*}])$.*

Proof. We actually construct such a cut. Let A^* be the set of vertices that are reachable from s in G_f (so $\overline{A^*}$ is the set of vertices not reachable from s i.e. there is no $s-x$ path in G_f for any $x \in \overline{A^*}$).

1. First, we show that $[A^*, \overline{A^*}]$ is a $s-t$ cut. since s is reachable form itself, $s \in A^*$, and by the hypothesis there is no $s-t$ path in G_f $t \in \overline{A^*}$. Hence $[A^*, \overline{A^*}]$ is a $s-t$ cut.

2. Second, we show that $size(f) = c([A^*, \overline{A^*}])$.

3. To see this we show that for any edge $e = xy$, where $x \in A^*$ and $y \in \overline{A^*}$, $f_e = c_e$.

4. Actually, if $f_e < c_e$, then by construction of G_f, there will be a forward edge $xy \in G_f$, such that the residual capacity of xy is $f_e - c_e > 0$. But then y should be in A^* as there is an $s-y$ path in G_f via x.

5. Similarly, for any edge $e = uv$, with $u \in \overline{A^*}$ and $v \in A^*$, we have $f_e = 0$.

6. Again, if $f_e > 0$, then by construction in the residual graph there will be a reverse edge vu with residual capacity equal to $f_e > 0$. And we get a $s - u$ path in G_f, contradicting the assumption that $u \in \bar{A}^*$.

7. Hence, all edges outgoing form A^* are completely saturated (no capacity left) and all edges incoming to A^* are completely unused.

8. Now using equation (1) we have

$$\begin{aligned} size(f) = f^{out}(A^*) - f^{in}(A^*) &= \sum_{e \text{ outgoing from } A^*} f_e - \sum_{e \text{ incoming to } A^*} f_e \\ &= \sum_{e \text{ outgoing from } A^*} c_e - \sum_{e \text{ incoming to } A^*} 0 \\ &= \sum_{e \text{ outgoing from } A^*} c_e - 0 \\ &= c([A^*, \overline{A^*}]) \end{aligned}$$

□

Theorem 10. *If f is a flow with a corresponding cut $[A^*, \overline{A^*}]$ such that $size(f) = c([A^*, \overline{A^*}])$, then f is a maximum flow and $[A^*, \overline{A^*}]$ is a minimum cut.*

Proof. We get this as an immediate corollary to Lemma 2. Because if there is flow of larger size than f, then that size of that flow is larger than the cut $c([A^*, \overline{A^*}])$ a contradiction to Lemma 2. Similarly if there is a cut of smaller capacity than $[A^*, \overline{A^*}]$, then that cut also contradicts Lemma 2. □

Theorem 11. *The Ford-Fulkerson algorithm returns a maximum flow.*

Proof. Since the Ford-Fulkerson algorithm returns a flow f such that G_f has no $s - t$ path. By Theorem 9 there is a cut with capacity equal to $size(f)$. Hence by Theorem 10 f is optimal. □

Lemma 12. *Given a maximum flow f in a network G, we can compute a minimum $s - t$ cut in G in time $O(m)$.*

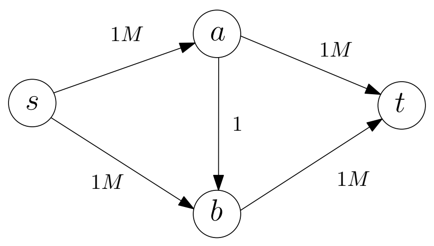
Proof. This minimum cut is given as a bonus. Just run a DFS or BFS in G_f to find the set of vertices reachable from s. Together with it's complement by the above theorem this is a min-cut. G_f can be computed in $O(n + m) = O(m)$, as we just have to make (at most) 2 edges for each edge in G. Now since G_f has at most $2m$ edges, a BFS in G_f takes at most $O(n + m) = O(m)$ time (of course if a max-flow f is given). □

2.9 Ford-Fulkerson Algorithm - Slow Example

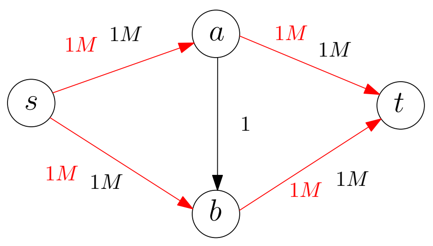
Recall the upper bound on the running time of $O(mC_s)$ the Ford-Fulkerson algorithm, where C_s is the capacity of the cut $[\{s\}, \overline{\{s\}}]$. Note that any cut is only an upper bound size of the maximum flow, in case when C_s is much greater than size of the maximum flow, a tighter upper bound is given by $O(m \cdot size(f))$, when f is a maximum flow. This can be proved using exactly the same argument as above.

Clearly, f can be arbitrarily large (e.g. 2^n or even super exponential in the size of input), hence this algorithm can be arbitrarily bad. The question is that this is only an upper bound, is it ever achieved meaning are there instances where this bound is actually tight. We give such an instance, where this indeed happens.

Consider the example in Figure 14 with 4 vertices and 5 edges. $1M$ is one million. The maximum flow clearly is of size $2M$. A few iterations of one possible execution are shown in Figure 15.



(a) Given network



(b) Maximum flow of size $2m$

Figure 14: Example network on which Ford-Fulkerson may run too slow

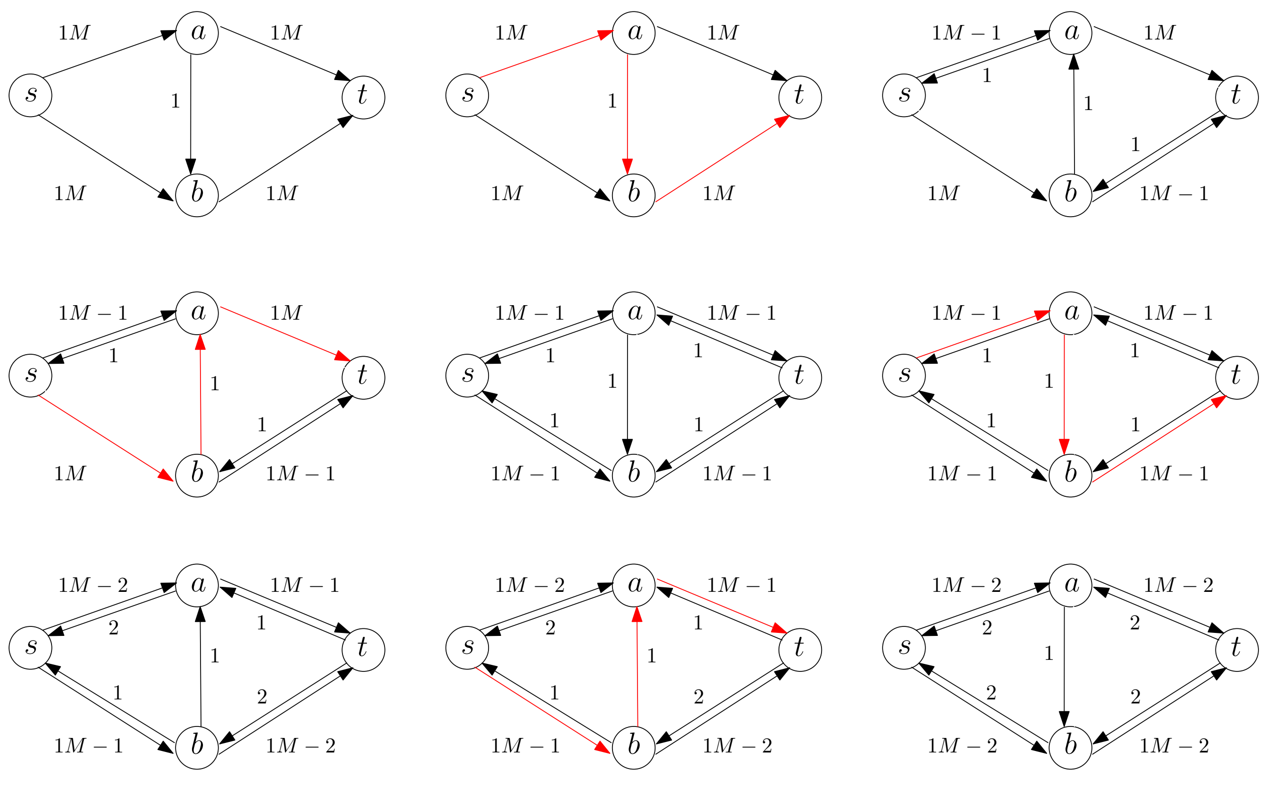


Figure 15: Few Iterations of Ford-Fulkerson on example in Figure 14

On this example the Ford-Fulkerson algorithm could potentially take $2M$ iterations, as in each step it augments 1 unit of flow to the existing flow. Fortunately, Ford-Fulkerson algorithm doesn't restrict the choice of augmenting path. We can choose the augmenting path wisely to fix the above problem.

3 Edmonds-Karp Algorithm

The only fix needed in the Ford-Fulkerson algorithm is to always choose an augmenting path that is of shortest length, i.e. the hop length (number of edges on the path) is as small as possible. This is the Edmonds-Karp Algorithm.

Note that an augmenting path in a residual graph is found while ignoring the weights (capacities). But we don't have to ignore the number of edges. This essentially amounts to using BFS to find a $s-t$ path in the augmenting graph rather than DFS.

Correctness of the Edmond-Karp algorithm follows immediately from the above proof, as that works for any augmenting path. Before analyzing its running time we run this algorithm, on the pathological example from Figure 14, as shown in Figure 16.

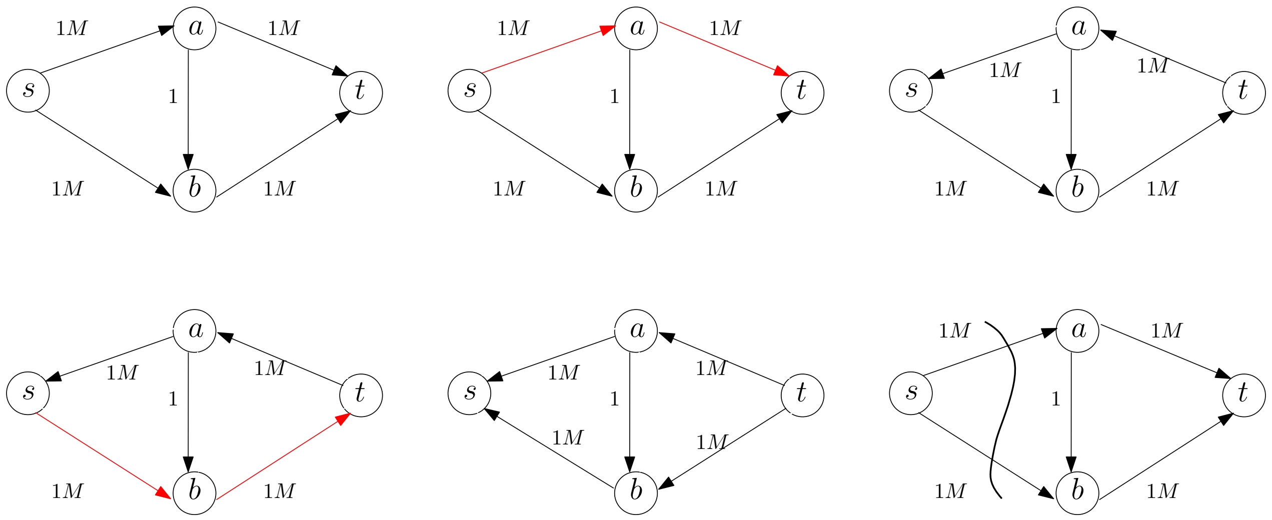


Figure 16: Running Edmond-Karps efficiently on example network in Figure 14

In any augmenting path algorithm, each augmentation saturates at least one edge on the augmenting path, which actually makes that edge unavailable for the next iteration (possibly many), at least in that direction. The key in this analysis is to realize that an edge is not saturated too many times. This was the fundamental problem in the pathological example as in Figure 15 the middle edge (a, b) was saturated too many times.

We need the following crucial lemma for proving a bound on the number of times an edge is saturated.

Lemma 13. *After each iteration of the Edmonds-Karp algorithm, the $d_{G_f}(s, v)$ (distance (hop-length) in G_f from s to v) does not decrease for all vertices v. Same is true for $d_{G_f}(v, t)$ and hence $d_{G_f}(s, t)$.*

Proof. This is a straight-froward observation, as when an edge (x, y) is saturated, in the residual network we remove the edge (x, y) (because its residual capacity is 0 (the edge (y, x) is there though. Let f be the flow such that the edge (x, y) is saturated in G_f. Let f' be the resulting flow. We know that $d_{G_f}(s, y)$ is greater than $d_{G_f}(s, x)$, by the sup-path optimality. i.e. if there was a shorter path from s to y, then we will use that shorter path to go to t, since we chose a shortest path from s to t. Now in $G_{f'}$, since the edge (x, y) is not there, there cannot be a shorter path from s to y (a backward edge can never decrease a distance, as it is from a vertex farther away from s (namely y) to a vertex closer to s (namely x).

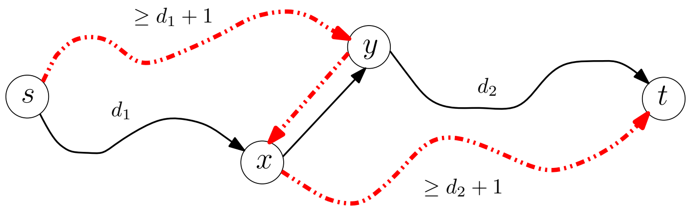
In other words note that distances from s to v are originally defined in $G = G_{f=0}$. All other edges added in subsequent residual networks are from vertices farther from s to vertices closer to s (as we only introduce reverse edges along a shortest path). $\square$

Using this we show when can the algorithm reuse an edge that is once saturated.

Lemma 14. *When an edge $e = (x, y)$ is saturated once, it cannot be reused until $d_{G_f}(s, t)$ strictly increases.*

Proof. Suppose when the edge $e = (x, y)$ is saturated via the path $P = s, \dots, x, y, \dots, t$. Since P is a shortest path, the length of P , $l(P) = d(s, t)$. Let us denote lengths of the subpath of P from s to x by d_1 and length of the subpath of P from y to t by d_2. In other words $d(s, t) = l(P) = d_1 + 1 + d_2$. Note that this also gives us that $d(s, y)$ at this stage is $d_1 + 1$.

Suppose the edge (y, x) is used at some point while working with $G_{f'}$ (since the edge (x, y) doesn't exist any more, for (x, y) to be used again, (y, x) must be used before it). By the previous lemma $d_{G_{f'}}(s, y) \geq d_1 + 1$, and $d_{G_{f'}}(x, t) \geq d_2 + 1$. If a path in $G_{f'}$ uses (y, x), since it is a shortest path, then we must have $d_{G_{f'}}(s, t) \geq d_1 + 1 + 1 + d_2 + 1 \geq d_1 + d_2 + 3 \geq l(P) + 2$.



Solid black line is P, when (xy) is first saturated
Red dashed lines is the path when edge (yx) is used the next time

Figure 17: An edge can only be saturated a limited number of times by Edmond-Karp Algorithm

So if the algorithm is going to use the edge (y, x), then distance from s to t must have increased. □

Theorem 15. *The Edmonds-Karp algorithm takes $O(nm^2)$ time.*

Proof. Since distance from s to t can only increase $O(n)$ times (maximum path length is $n - 1$). So for a fixed value of distance $d(s, t)$, a given edge (x, y) can be saturated only once. After (x, y) is saturated once, If the distance does not increase, then the algorithm can saturate other edges but not (x, y). Hence for this fixed value of distance there can be at most $O(m)$ iterations, as in every iteration some edge must be saturated.

After which the distance must increase, and for this second fixed value again there could be $O(m)$ iterations after which there must be an increase in distance.

Since the total number of different values for $d(s, t)$ is $O(n)$, in total there could be $O(nm)$ iterations. Recall that each iteration takes $O(n + m) = O(m)$ (BFS), the total running time of this algorithm is bounded above by $O(nm^2)$. □

One point to note is that this could potentially be $O(n^5)$ for very dense graphs. But at least it is polynomial in the size of the input. The upper bound on runtime does not directly depend on the values of the capacities (and hence in the size of the flow), but different values of capacities will effect the running time. To appreciate this point suppose all capacities are 0 in this case there will be no iterations. Similarly if the all capacities are the same, then a given path will saturate many edges at once, hence there will be fewer iterations.

There are many better algorithms. You are encouraged to read about them, especially, the push-relabel algorithm in your textbook. Some algorithms are directly based on the Ford-Fulkerson algorithm (like this Edmonds-Karp algorithm). You are particularly encouraged to read about the scaling algorithm. This is a very useful trick. The state of the art algorithm for Max-Flow algorithm is $O(nm)$.

4 Further Reading

You should read on the following and know them on your own.

1. What if there are multiple sources and/or sinks.? This is dealt with very easily by introducing a dummy source (and/or a dummy sink) node. This dummy source is connected to each original source s_i by an edge with capacity equal to $c([\{s_i\}, \{\bar{s_i}\}])$. If it is not clear why this should be the capacity of this edge, think about it until it is clear.

2. What if capacities are not integral? Then, the FF algorithm could be arbitrarily slow. Rational numbers can be handled by multiplying all numbers with their LCM but real numbers are trouble. We need to use a non-augmenting paths based algorithm.

3. The various variants of maximum flow, in particular the circulation problem. There are a lot of applications of the maximum flow problem and its variants. Please go through the book chapter and read about the following applications

   - Bipartite Matching
   - Image Segmentation
   - Survey Design
   - Disjoint Path
   - Project Selection
   - Baseball Elimination