MATHEMATICAL EXPLANATIONS

SIDDHANT CHAUDHARY

Abstract. In this document, we will mathematically explore all of the concepts being used.

Contents

1. Non-Linear Data Structures

1.1. **Binary Indexed Trees.** *Fenwick Trees* are used to solve the problem of dynamic range sum queries efficiently. They are useful because they are much simpler to code and much faster than segment trees. Let's begin with a lemma.

Lemma 1.1. *For any integer type* `i`, `i&(-i)` *produces the least significant bit in* `i`.

Proof. Note that negative integers are stored using the two's complement. So, if we have an N-bit positive integer x, $-x$ will have the binary representation of $2^N - x$ (equivalently, one flips the bits of x, and adds a 1 to obtain the representation of $-x$). Now, suppose the least significant bit of x is l steps from the right. This means that all positions $0, 1, ..., l-1$ steps from the right are 0s in the binary representation of x. So, if we flip the bits of x, all these positions will be 1. Adding a 1, we see that the bit at position l is set to 1, and positions $0, ..., l-1$ from the right are all 0s. Taking the and, we see that the result has only a 1 at position l, proving the claim. ■

Moving on, we now describe the Fenwick Tree. Suppose we have an array $A[1, ..., n]$ (we use one-based indexing). Let $FT[1, ..., n]$ be another array; $FT[i]$, for any i, will

Date: December 2019.

store the sum of all elements in the range $[i - LS(i) + 1, i]$, where $LS(i)$ is the *least-significant bit* of i. For example, if i is a power of 2, then $LS(i) = i$, and hence $FT[i]$ will be the sum

$$FT[i] = \sum_{k=1}^{i} A[k]$$

Here is the nice trick: given an index b, we want to compute

$$\sum_{k=1}^{b} A[k]$$

To do this, we compute the values $\{FT[b_0], FT[b_1], ..., FT[b_k]\}$. Here, $b_0 = b$ and $b_i = b_{i-1} - LS(b_{i-1})$ for each suitable i. In simple words, starting with b, we strip off the least significant bit one by one. We only need to do this for $O(\log b)$ many steps. Using this, given the function FT is *good enough* (for example, here FT is the sum function, which has many nice properties), we can compute range queries in logarithmic time.

Let's see how this structure handles update queries. Suppose we want to update the element at index k. So, we want to change $A[k]$ to some new value. So, we will have to update $FT[i]$ for all those indices i which include the index k in their summation. In other words, we want to update all i such that

$$i - LS(i) + 1 \leq k \leq i$$

Then, by doing casework on $LS(i)$, we see that all such i are obtained by the sequence $\{c_0, ..., c_k\}$, where $c_0 = k$ and $c_{i+1} = c_i + LS(k)$. In other words, starting from k, we add least significant digits (this is one of the beautiful connections between these operations). So again, we only need to update $O(\log n)$ many values.

2. Flow Networks

2.1. **Definitions and the Set up.** In this subsection, we will formally define flow networks and prove some of their useful properties.

Definition 2.1. A *flow network* is a graph $G = (V, E)$ with two distinguished vertices s and t, called the *source* and *sink* vertices respectively. Each edge $(u, v) \in E$ has a non-negative capacity, denoted by $C(u, v)$. If $(u, v) \notin E$, we say $C(u, v) = 0$.

Definition 2.2. A *flow* on a flow network G is a function $f : V \times V \to \mathbf{R}$ that satisfies the following constraints.

(1) For all $u, v \in V$, $0 \leq f(u, v) \leq c(u, v)$. This is called the *capacity constraint.*
(2) For all $u, v \in V$, $f(u, v) = -f(v, u)$. This is called the *skew symmetry constraint.*
(3) For all $u \in V - \{s, t\}$, we have

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

This is the so called *flow conservation law.*

Also, the following assumptions are made about the network G.

- There are no self-loops in G, i.e $f(u, u) = 0$ for every $u \in V$.

- There are no *two-cycles* in G; formally, there is no pair u, v of vertices in G such that both (u, v) and (v, u) is an edge in G. Note that this is *enforced* by property (2) above, since we are assuming that flows along edges are non-negative. The reason to make this assumption is for **(i)** simplification and **(ii)** networks G which have two-cycles can be converted to an equivalent network G' which have no two-cycles; intuitively, this is achieved by adding a new vertex and creating a three cycle. Here, *equivalence* of networks means that the *max-flow* (which we will define in a moment) in the two networks will be the same.

Definition 2.3. Let A, B be any arbitrary subsets of V. Define the quantity $f(A, B)$ as follows.

$$f(A, B) := \sum_{u \in A, v \in B} f(u, v)$$

We call this quantity the *net flow* going from A to B. The reasoning behind this naming will be proved later.

Definition 2.4. The *value* of a flow f, denoted by $|f|$, is defined to be

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

Informally, this represents the net flow coming out of s, i.e the flow coming out of s minus the flow going into s. This is a good definition of *net flow*, as it will turn out that the net flow coming out of s is equal to the net flow going into t.

Proposition 2.1. *Let f be any flow, and let X, Y and Z be arbitrary subsets of V. Then the following hold.*

(1) $f(X, X) = 0$.
(2) $f(X, Y) = -f(Y, X)$.
(3) $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ *provided* $X \cap Y = \phi$.

Proof. Let us begin with (1). There is nothing more to this than simply expanding things out.

$$\begin{aligned} f(X, X) &= \sum_{u \in X, v \in X} f(u, v) \\ &= \sum_{u \in X} f(x, x) + \sum_{u \in X, v \in X, u \neq v} f(u, v) \\ &= 0 + \sum_{u \neq v} f(u, v) + f(v, u) \\ &= 0 + \sum_{u \neq v} f(u, v) - f(u, v) \\ &= 0 \end{aligned}$$

Property (2) is immediate by the definition of net flow. Property (3) is also trivial and follows from the definition. ∎

Theorem 2.2. $|f| = f(V, t)$. *Informally, the flow coming out of the source is equal to the flow going into the sink.*

Proof. To begin with, consider the sets $\{s\}$ and $V - \{s\}$, which are clearly disjoint. So,

$$f(V, V) = f(s, V) + f(V - \{s\}, V)$$

The left hand side is clearly zero. So, we get

$$f(s, V) = -f(V - \{s\}, V) = f(V, V - \{s\})$$

Now again, consider the sets $\{t\}$ and $V - \{s, t\}$, which are clearly disjoint. So, we have the following.

$$\begin{aligned} f(V, V - \{s\}) &= f(V, t) + f(V, V - \{s, t\}) \\ &= f(V, t) - f(V - \{s, t\}, V) \\ &= f(V, t) - \sum_{u \in V - \{s,t\}, v \in V} f(u, v) \\ &= f(V, t) + 0 \end{aligned}$$

where in the last step, we have used flow conservation law. ■

Definition 2.5. For a flow network G, an (S, T) *cut* of a flow network G is a partition of V such that $s \in S$ and $t \in T$. If f is a flow of G, then the flow across the cut $f(S, T)$. The *capacity* of a cut, denoted by $C(S, T)$, is defined as follows.

$$C(S, T) := \sum_{u \in S, v \in T} c(u, v)$$

Clearly, the flow across a cut cannot exceed the capacity of the cut, i.e

$$f(S, T) \leq C(S, T)$$

2.2. **Flow value with Cuts.** We begin by proving a simple lemma.

Lemma 2.3. *For any flow f and any cut (S, T), we have*

$$|f| = f(S, T)$$

Informally, this is true because the source is on one side of the cut, while the sink is on the other side.

Proof. The proof is straightforward.

$$\begin{aligned} f(S, T) &= f(S, V) - f(S, S) \\ &= f(S, V) + 0 \\ &= f(s, V) + f(S - \{s\}, V) \\ &= |f| \end{aligned}$$

as the second value is zero by conservation of flow. ■

Corollary 2.3.1. *If (S, T) is any cut, then $|f| \leq C(S, T)$. Hence, $|f|$ is less than the minimum capacity over all the cuts.*

2.3. **Residual Networks.** Let $G(V, E)$ be a flow network. For any pair of vertices (u, v), define the *residual capacity* $c_f(u, v)$ as

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

The *residual network* is the graph $G_f(V, E_f)$, where E_f is defined to be the set of all edges with *positive* residual capacity.

Definition 2.6. An *augmenting path* is any path from s to t in the residual network G_f. The *residual capacity* of an augmenting path is the minimum value of any edge along the path. Formally,

$$c_f(p) := \min_{(u,v)\in p} c_f(u,v)$$

2.4. **How to augment a flow?** Suppose we are given a flow f, and suppose we have an augmenting path p in residual network. The question is: using this augmenting path, can we somehow increase the flow? The answer is yes, and this is what we will prove in this section.

Define a flow f_p on the network G_f as follows.

$$f_p(u,v) = \begin{cases} c_f(p) & , \quad \text{if } (u,v) \in p \\ 0 & , \quad \text{otherwise} \end{cases}$$

That f_p is really a flow on G_f is not hard to see. Intuitively, we are just sending in the maximum possible flow that we can send along the path p, and nowhere else.

Next, we define a flow $f + f_p$ on the original network G as follows.

$$(f+f_p)(u,v) = \begin{cases} f(u,v) + f_p(u,v) - f_p(v,u) & , \quad \text{if } (u,v) \in E \\ 0 & , \quad \text{otherwise} \end{cases}$$

Try to prove that this is indeed a flow in the original network. Infact, it is also true that

$$|f + f_p| = |f| + |f_p|$$

and hence this increments are flow.

2.5. **Max-Flow Min-Cut Theorem.** In this section, we will prove a very important theorem, which will enable us to devise a max flow algorithm.

Theorem 2.4 (Max-Flow Min-Cut Theorem)**.** *The following are equivalent.*

(1) $|f| = C(S,T)$ *for some cut* (S,T).
(2) f *is a maximum flow.*
(3) f *admits no augmenting paths.*

Proof. First, let us show that (1) $\implies$ (2). But this is immediate from **Corollary** 2.3.1; since $|f| = C(S,T)$ for some cut (S,T), f has to be a max flow, because it can't be increased. This proves the claim.

Let us now consider (2) $\implies$ (3). Suppose f is a max flow. If f admits an augmenting path p, then clearly we can increase f by $c_f(p)$ and get a strictly greater flow. So, there cannot be any augmenting paths.

Finally, let us prove (3) $\implies$ (1), and this will be a little more involved, but the argument is intuitive enough. Suppose f admits no augmenting paths, i.e in the residual network G_f, there are no paths from s to t. With this in mind, define the set S as follows.

$$S := \{u \in V \mid u \text{ is reachable from } s \text{ in } G_f\}$$

Then define $T := V - S$, and clearly (S,T) is an $s-t$ cut. We will show that

$$f(S,T) = C(S,T)$$

and as above in (1) $\implies$ (2), this will finish our proof. The argument is intuitive; suppose $u \in S$ and $v \in T$. Clearly, the edge (u,v) cannot be a part of the residual

network G_f, by the very definition of the cut. This means that $c_f(u, v) = 0$, which in turn means

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

Clearly, by the definition of $f(S, T)$ and $C(S, T)$, this means that

$$f(S, T) = C(S, T)$$

and hence the proof is complete. ■

2.6. **Edmonds-Karp Algorithm.** The *Edmonds-Karp* algorithm is very similar to the naive *Ford-Fulkerson* algorithm, but there is a key difference in how the Edmonds-Karp algorithm finds the augmenting path. In this section, we will analyse the Edmonds-Karp algorithm, and if not specified, the flows/capacities will be assumed to be integral.

The Edmonds-Karp algorithm can be described as follows.

(1) Given G, compute G_f, the residual network.
(2) Find a shortest path from s to t in the residual network; this can be done using BFS. Here by *shortest* we mean that each edge was weight 1.
(3) Having found a shortest path, augment the flow on that path. Get a new flow.
(4) Repeat the algorithm from step (1).

Suppose T is the number of steps for which the algorithm runs. Steps (2) and (3) take $O(E)$ time, which is trivial. So, the running time is $O(TE)$. We will now show that $T = O(VE)$.

Definition 2.7. Given a residual network G_f, define $\delta_f(v)$ to be the length of the shortest path from s to v, for $v \in V$.

Lemma 2.5 (Monotonic)**.** *$\delta_f(v)$ does not decrease during the algorithm execution, for all $v \in V$. So, this quantity can only increase or stay the same.*

Proof. We prove this by contradiction. Suppose f' is the new flow obtained from f after a single step of the algorithm. Also, suppose there is some $v \in V$ such that

$$\delta_{f'}(v) < \delta_f(v)$$

Among such violating vertices, let ν be the vertex such that $\delta_{f'}(\nu)$ is minimum. So, there is some path $s \rightsquigarrow \nu$ in the residual graph $G_{f'}$, and suppose the path is $s \rightsquigarrow u \to \nu$, i.e u is the predecessor of ν in this path. Clearly, we have

$$\delta_{f'}(\nu) = \delta_{f'}(u) + 1$$

Also, by the definition of ν, we have that

$$\delta_{f'}(u) + 1 \geq \delta_f(u) + 1$$

giving us the inequality

$$\delta_{f'}(\nu) \geq \delta_f(u) + 1$$

Now, we will handle two cases.

(1) In the first case, suppose $(u, v) \in G_f$. In that case, by the triangle inequality of shortest paths, we have

$$\delta_f(u) + 1 \geq \delta_f(\nu)$$

which gives us the inequality

$$\delta_{f'}(\nu) \geq \delta_f(\nu)$$

which is a contradiction to our assumption.

(2) In the second case, we have $(u, \nu) \notin G_f$. Also, we clearly know that $(u, \nu) \in G_{f'}$. Clearly, this is only possible if $(\nu, u) \in G_f$ (and in fact this edge lies on the augmenting path we found to get f'). In this case, because Edmonds-Karp only augments shortest paths, we have

$$\delta_f(u) = \delta_f(\nu) + 1$$

and this again is a contradiction to our assumption.

■

Theorem 2.6. *The number of iterations in the Edmonds-Karp algorithm is $O(VE)$.*

Proof. Let the minimum weight edge on an augmenting path be called the *critical edge* of the path. We will show that each edge of G can be critical atmost $O(V)$ times, and this will complete the proof.

Suppose the current flow is f, and we are augmenting a path that has critical edge (u, v) (note that either $(u, v) \in E$ or $(v, u) \in E$, since residual graphs only have edges with positive flow). Suppose the new flow after augmenting the flow f is f'. Since we are augmenting the edge (u, v) of G_f, the new residual network $G_{f'}$ won't contain the edge (u, v); to prove this, we consider the following two cases.

(1) In the first case, suppose $(u, v) \in E$. Here, since $(u, v) \in G_f$ and is the critical edge of the augmenting path, after augmentation, the flow on (u, v) will be *saturated*, i.e it will reach its full capacity. Hence, in the new residual $G_{f'}$, the edge (u, v) won't be present.

(2) In the second case, suppose $(u, v) \notin E$. The only way (u, v) can be in G_f is when $(v, u) \in E$ and $f(v, u) > 0$. After augmenting this path, the new flow on the edge (v, u) will become

$$f'(v, u) = f(v, u) - f_p(u, v) = 0$$

where above p is the augmenting path. So, in the new residual graph $G_{f'}$, edge (u, v) won't be present.

Since we are augmenting the edge (u, v) on the flow f, we have

$$\delta_v(f) = \delta_u(f) + 1$$

Now, suppose the edge (u, v) is critical *again* for some flow f''. Since f', the flow immediately after f, does not contain the edge (u, v), the only way for f'' to contain the edge (u, v) again is when the edge (v, u) was augmented at some point before the flow f'' was obtained. Suppose f''' is the flow after f and before f'' which augmented the edge (v, u). So,

$$\delta_u(f''') = \delta_v(f''') + 1 \geq \delta_v(f) + 1 = \delta_u(f) + 2$$

where above we have used the monotonicity of δ_f. This proves that between two instances of (u, v) to be a critical edge, $\delta(u)$ increases by atleast 2. Since this value is bounded above by $|V|$, we see that each such edge can be a critical edge atmost $|V|/2 = O(V)$ times. Hence, over all edges, the number of iterations is atmost $O(VE)$, and this proves the claim. ■

2.7. **Bipartite Matching.** We can use bipartite matchings to solve the *maximum bipartite matching problem*. Suppose we have a bipartite graph $G = (V, E)$. Our goal is to find a *maximum matching* on this graph. We will see how to frame this problem as a max flow problem on the graph. The algorithm is quite simple, and we now describe it.

Suppose the graph is partitioned into sets L, R (L is the set of the vertices in the left partition, and similarly R is the set of the vertices in the right partition). We create a corresponding flow network G' as follows: introduce new vertices s and t. For each $u \in L$, create a directed edge (s, u). For each $v \in R$, create a directed edge (v, t). Assign a capacity of 1 to all the edges.

Then, any *integer valued* flow in this graph corresponds to a matching. This is not hard to show (try to do it as an exercise). Moreover, if we use Ford-Fulkerson on this graph, all flows obtained in the intermediate steps will be *integer values*; this is the so called *integrality theorem*, and this is not hard to show either (intuitively, at each step, all updates are integer valued, assuming the flow and the capacities are integers). Moreover, it is true that the max flow in this graph is equal to the cardinality of the maximum matching in the graph (again, try to prove all these facts).

It follows that Ford-Fulkerson will give us the maximum matching. Note that a maximum matching can have size atmost $O(V)$, and hence Ford-Fulkerson runs in time $O(VE)$ on such a graph.

2.8. **Dinic's Algorithm and Blocking Flows.** A good resource for learning this is the following: http://timroughgarden.org/w16/l/l2.pdf

Now, we will explore a faster max flow algorithm, namely *Dinic's Algorithm*. This algorithm runs in time $O(V^2E)$ instead of the usual $O(VE^2)$, and clearly this is a faster algorithm.

The algorithm is pretty similar to Edmonds-Karp.

(1) We start with a zero flow, i.e all edges have flow equal to 0.
(2) While there is an $s-t$ path in the residual graph G_f, we construct the *layered graph* L_f (which will be defined in a moment) from the residual graph G_f via BFS.
(3) We'll then compute the *blocking flow* g in the layered graph L_f (again, we'll define this in a moment), and we augment the flow f by g (we'll see how to compute g).

Remark 2.6.1. Note that we are computing the blocking flow in the graph L_f (the layered graph) and not G_f; this is a really important but subtle thing to note.

Definition 2.8. Let G be a flow network with a given flow f. Let G_f be the residual graph. Assume that there is an $s-t$ path in G_f. The *layered graph* L_f is defined as follows: in the residual graph G_f, run a BFS from the source vertex s until the vertex t is reached. Let $\delta(s,t)$ denote the size of the *shortest path* from s to t. Then, for each $0 \leq i \leq \delta(s,t)$, define the ith *layer* to be the collection of all those vertices which are at a shortest distance of i from s. Then, define L_f to be the subgraph of G_f consisting only of those edges of G_f which go from layer j to layer $j+1$ for some j (so we are not considering edges within the same layer and edges that go back).

So, the layered graph L_f can really be thought of as *layers* of vertices; the first layer consist of those vertices in L_f that are at a distance of 1 from s, and similarly for other layers. Clearly, all *shortest paths* from s to t in graph G_f are paths from s to t in L_f. Next, let us define *blocking flows*.

Definition 2.9. A *blocking flow* g on a network G is a flow such that, for every $s-t$ path p of G, some edge of p is saturated by g, i.e $f(u,v) = c(u,v)$ for some edge (u,v) in G. So intuitively, a *blocking flow* cancels out atleast one edge in *every* $s-t$ path.

Just like in the proof of Edmonds-Karp, we will prove an assertion about shortest paths increasing if we augment our flow by a *blocking flow*. First, we will prove a simple lemma.

Lemma 2.7. *Let $d(i)$ denote the shortest distance of vertex i to the source s in the graph G_f. Then, d is a valid distance labelling function on G_h; i.e, for all edges $(i,j) \in G_h$, we have*

$$d(j) \leq d(i) + 1$$

Proof. Suppose (i,j) is an edge of G_h. Clearly, either $(i,j) \in G_f$ or $(i,j) \notin G_f$. We will handle both of these cases.

(1) First, suppose $(i,j) \in G_f$. In this case, there is nothing to prove, because $d_j \leq d_i + 1$ in this case by the definition of shortest paths.
(2) In the second case, we have that $(i,j) \notin G_f$. Then, the only way (i,j) is introduced in G_h is if we sent some flow on the edge (j,i) in the previous iteration, i.e $(j,i) \in G_f$. Since g is a blocking flow on L_f, this clearly means that $(j,i) \in L_f$, which in turn means that

$$d(j) = d(i) - 1 \leq d(i) + 1$$

■

Proposition 2.8. *Let G be a flow network, and let f be a flow on G. Consider the graph G_f, the residual graph. Let $d(f)$ denote the shortest distance between s and t in G_f. Let g be a blocking flow in L_f (the layered graph). If $h = f + g$, i.e if flow h is obtained by augmenting f by g, then $d(h) > d(f)$. So, by augmenting a flow with a blocking flow, the shortest $s - t$ distance strictly increases.*

Proof. Let d be the distance from the source s in G_f, and let d' be the distance from the source s in G_h. We will show that

$$d(t) < d'(t)$$

i.e after every iteration, the shortest $s - t$ distance strictly increases.

First, let P be *any* $s - t$ path in the residual graph G_h. We claim that there is some edge $(u,v) \in P$ in the residual graph G_h such that either $(u,v) \notin G_f$ or $d(v) \neq d(u)+1$. We now prove this. Suppose all the edges in the path P are *old* edges, i.e if $(u,v) \in P$, then $(u,v) \in G_f$. For the sake of contradiction, suppose all the edges $(u,v) \in P$ satisfy $d(v) = d(u) + 1$. Then, it is clear that P is actually a shortest $s - t$ path in G_f; moreover, all the edges of P are actually edges in the layered graph L_f. Now, since g is a *blocking flow* on the layered graph L_f, there is some edge $(u,v) \in P$ that is completely saturated. But, this would imply that, after augmenting f by g (to obtain h), the edge (u,v) won't be present in the new residual graph, i.e $(u,v) \notin G_h$; this is a contradiction.

We now prove the main statement of the proposition, i.e $d(t) < d'(t)$. Again, let P be any $s - t$ path in G_h. So, as we've shown above, there is some edge $(u,v) \in P$ such that either $(u,v) \notin G_f$ or $d(v) \neq d(u) + 1$. In the first case, i.e $(u,v) \notin G_f$, the only way the edge (u,v) is present in G_h is if we have sent some flow on the edge (v,u); clearly, this implies that $(v,u) \in L_f$, which means that $d(v) < d(u)$. In the other case, namely that $d(v) \neq d(u) + 1$, we clearly have that $d(v) \leq d(u)$ since by **Lemma** 2.7, we have that $d(v) \leq d(u) + 1$. In particular, we have shown that for any such path P, there is some edge $(u,v) \in P$ such that $d(v) \leq d(u)$. This clearly implies that $|P| > d(t)$, and hence this means that $d'(t) > d(t)$, completing the proof. ■

Corollary 2.8.1. *Dinic's algorithm takes atmost $O(V)$ iterations to run, and the time complexity of the algorithm is $O(V \cdot BF)$, where BF is the time required to compute the blocking flow.*

Proof. This is clear; because the shortest $s - t$ distance is strictly increasing at after every iteration, there can be atmost $O(V)$ iterations since the maximum possible shortest distance is $O(V)$. The second statement clearly follows from the first. ■

Remark 2.8.1. It follows that solving the blocking flow problem efficiently will give us a good algorithm to compute max flows. In the upcoming sections, we will see how to compute the blocking flow in $O(VE)$ time, giving us $O(V^2E)$ running time for Dinic's Algorithm.

2.9. **Computing Blocking Flows Efficiently.** Now we will see how to compute blocking flows efficiently in the layered graph L_f. The idea is simple: we will start a DFS from the source s. If at any point of time we reach t, we know that we have found a shortest $s - t$ path (since we are working in the graph L_f); we will then take the smallest capacity edge on this path, and we will augment the flow along this path. We will then repeat the same procedure, and we will make sure *not to repeat edges* (which will be clear in the pseudocode). This is important because if we repeat edges, the time complexity wouldn't be $O(VE)$. To achieve this, for each vertex we will remember a *pointer* which will store information about which edge of the vertex to process next. Please have a look at the pseudocode.

Convince yourself that computing a blocking flow using the given procedure takes time $O(VE)$; as a hint, note that each augmentation causes an edge in L_f to be completely saturated.

Algorithm 1 Computing Blocking Flows

Input: Network G, layered graph L_f (computed using BFS).
For each v, degree$[v]$ denotes the number of edges from v to the next level in L_f.
$N[v][i]$ denotes the ith neighbor of vertex v, where $0 \leq i <$ degree$[v]$.
$C[v]$ denotes the index of *current* neighbor of v. Current neighbor is $N[v][C[v]]$.
Res$[v][i]$ denotes capacity of the ith edge of v in L_f, computed beforehand. Note that these really are the residual capacities of the original edges in G. The result will be returned by decreasing these capacities.

flow $\leftarrow 0$.
for each vertex v **do**
  $C[v] = 0$
end for
while true do
  $f \leftarrow$ DFS(s)
  if f $= 0$ **then**
    return flow ▷ Return the value of the blocking flow
  else
    flow $\leftarrow$ flow $+ f$
    Augment the current $s - t$ path by f, i.e reduce all capacities on this
    path by f. The current path is just $s \rightarrow N[s][C[s]] \rightarrow \cdots \rightarrow t$.
  end if
end while
function DFS(v)
  ▷ Search a path from v to t of positive capacity.
  ▷ If there is no such path, return 0.
  ▷ As a side effect, this may increase $C[v]$.
  if $v = t$ **then**
    return ∞
  end if
  while true do
    if $C[v] =$ degree$[v]$ **then**
      return 0 ▷ v is *dead*
    end if
    if Res$[v][C[v]] = 0$ **then**
      $C[v] \leftarrow C[v] + 1$ ▷ Move to the next neighbor
    else
      $f \leftarrow$ DFS$(N[v][C[v]])$
      if $f > 0$ **then**
        return $\min(f,$ Res$[v][C[v]])$
      else
        $C[v] \leftarrow C[v] + 1$
      end if
    end if
  end while
end function

3. Push-Relabel Paradigm

3.1. **Preflow.** We will begin by defining a *preflow*.

Definition 3.1. A *preflow* on a flow network G is a function $f : V \times V \to \mathbf{R}$ that satisfies the capacity constraint, and also satisfies a relaxed version of the conservation constraint. Formally, for all $(u, v) \in E$, we have

$$0 \leq f(u, v) \leq c(u, v)$$

and also for every $u \in V - \{s\}$, it is true that

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

Ofcourse, we are still assuming that f is anti-symmetric. The above equation just means that the flow into a vertex could be more than the flow out of a vertex (except the source s). This quantity is also called the *excess flow* into vertex u. For a *preflow* f, the *residual network* G_f is defined as before.

Watch lecture from 10:30 mark.