9.2 Network Flow

We are given a directed graph with weighted edges, a source node, and a sink node. A *flow* is sent from the source to the sink. Each edge weight represents the maximum capacity of that edge. For every node besides the source and the sink node, total flow in is equal to total flow out. We can think of a flow network as a series of pipes through which water travels from an entrance to an exit in the network. The edge capacities represent pipe thickness. At any node, the total rate at which water enters the node must equal the total rate at which it exits the node, and along any path, the rate at which water flows is bottlenecked by the thinnest pipe.

More formally, for a graph $G(V, E)$, where $c(u, v)$ represents the capacity of the edge from u to v, the flow $f(u, v)$ from a source s to a sink t satisfies

$$\begin{aligned} f(u,v) &\leq c(u,v) && \forall (u,v) \in E \\ f(u,v) &= -f(v,u) && \forall (u,v) \in E \\ \sum_{v \in V} f(u,v) &= 0 && \forall u \in V \setminus \{s,t\} \\ \sum_{v \in V} f(s,v) &= \sum_{v \in V} f(v,t) = |f|, \end{aligned}$$

where $|f|$ represents the total flow from the source to the sink.

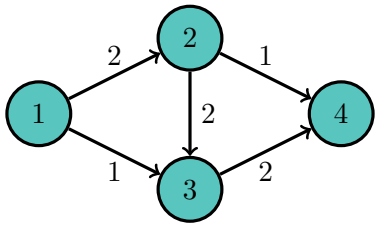
A *maximum flow* is one that maximizes the total flow $|f|$ from s to t, or in our example, maximizes the rate at which water can flow through our network.

We'll also define the residual capacity $c_f(u, v) = c(u, v) - f(u, v)$. Note that $c_f(u, v) \geq 0$ by the conditions imposed on f. The residual capacity of an edge represents how much capacity is left after a certain amount of flow has already been sent. We therefore have the residual graph $G_f(V, E_f)$, where E_f is the graph of residual edges, or all edges $(u, v) \in V^2$ satisfying $c_f(u, v) > 0$.

A natural approach to "solving" this problem would be to simply greedily add flow.

Find a path from the source to the sink in which all the edges have positive weight in the residual graph. Send flow along this path; that is, find the max flow across this path, which is the minimum weight of any edge on this particular path. Call this value *cap*. Then subtract *cap* from the residual capacity of every edge in the path. We repeat, and this is guaranteed to terminate since on any given move, we remove an edge from our residual graph.

What is wrong with our greedy approach? Consider the following graph:



The max flow from vertex 1 to vertex 4 is 3, but greedy gives only 2. This is because the best possible single path from the source to the sink may not be included the best possible overall flow.

9.2.1 Ford-Fulkerson

We somehow need a way to fix the inclusion of any suboptimal paths in our greedy approach, or to "send flow back" in case we sent it through a suboptimal path. We do this by introducing the *reverse edge* to our residual graph.

Find a path from the source to the sink in which all the edges have positive weight in the residual graph. Find the max flow across this path, which is the minimum weight of any edge on this particular path. Call this value *cap*. Then subtract *cap* from the residual capacity of every edge along the path and *increment the residual capacity of the reverse edge* (the edge connecting the same two vertices but running in the opposite direction) by *cap*. We call this operation on the path *augmenting the path*. We simply choose an augmenting path until no such paths exist.

Algorithm 10 Ford-Fulkerson

function AUGMENTPATH(path $p = \{v_i\}_{i=1}^{m}$, where $(v_i, v_{i+1}) \in E_f$, $v_1 = s$, $v_m = t$)
  $cap \leftarrow \min_{i=1}^{m-1}(c_f(v_i, v_{i+1}))$
  for $i \equiv 1, m-1$ **do**
    $f(v_i, v_{i+1}) \leftarrow f(v_i, v_{i+1}) + cap$
    $c_f(v_i, v_{i+1}) \leftarrow c_f(v_i, v_{i+1}) + cap$
    $f(v_{i+1}, v_i) \leftarrow f(v_{i+1}, v_i) - cap$
    $c_f(v_{i+1}, v_i) \leftarrow c_f(v_{i+1}, v_i) + cap$ ▷ incrementing reverse edge
function MAXFLOW($G(V, E)$, $s, t \in V$)
  for all $(u, v) \in V^2$ **do**
    $f(u, v) \leftarrow 0$
    $c_f(u, v) \leftarrow c(u, v)$
  $|f| \leftarrow 0$
  while $\exists p = \{v_i\}_{i=1}^{m}$, where $(v_i, v_{i+1}) \in E_f$, $v_1 = s$, $v_m = t$ **do**
    $cap \leftarrow \min_{i=1}^{m-1}(c_f(v_i, v_{i+1}))$
    $|f| \leftarrow |f| + cap$
    AUGMENTPATH(p)
  return $|f|$

The difference between this algorithm and the greedy approach from earlier is that the paths we now allow may run along a reverse path, essentially undoing any suboptimal flow from earlier. These more general paths in our residual graph are called *augmenting paths*.

This algorithm is guaranteed to terminate for graphs with integral weights. Its performance is bounded by $O(Ef)$, where f is the maximum flow and E is the number of edges, as finding a path from s to t takes $O(E)$ and increments the total flow f by at least 1. The concept of removing edges can't be used to produce a stricter bound because while an edge in one direction may be removed from the residual graph, doing so creates an edge in the other direction.

In its crudest form, Ford-Fulkerson does not specify on which path to push flow if multiple paths exist. It simply states that as long as such a path exists, push flow onto it. In addition to being slow, Ford-Fulkerson, as it is stated, is not guaranteed to terminate for graphs with non-integral capacities. In fact, it might not even converge to the maximum flow for irrational capacities. However, these problems can be fixed by simply specifying how the algorithm chooses the next path on which to

push flow. Nonetheless, the Ford-Fulkerson algorithm is formulated beautifully mathematically and such is useful from a math perspective, as we will see with the Max-Flow Min-Cut Theorem.

9.2.2 Max-Flow Min-Cut Theorem

On a graph $G(V, E)$, an *s-t cut* $C = (S, T)$ splits the partitions V into S and T satisfying $s \in S$ and $t \in T$. The *cut-set* of C is the set of edges $\{(u, v) \in E : u \in S, v \in T\}$. The *capacity* of an s-t cut is given by

$$c(S, T) = \sum_{(u,v)\in S\times T} c(u, v).$$

A *minimum cut* is an s-t cut that minimizes $c(S, T)$.

A cut represents a set of edges that, once removed, separates s from t. A minimum cut is therefore a set of edges that does this but minimizes the total capacity of the edges necessary to disconnect s from t.

The max-flow min-cut theorem states that the maximum value of any s-t flow is equal to the minimum capacity of any s-t cut.

First, the capacity of any cut must be at least the total flow. This is true by contradiction. Any path from s to t has an edge in the cut-set, so therefore any flow from s to t is upper bounded by the capacity of the cut. Therefore, we need only construct one flow and one cut such that the capacity of the cut is equal to the flow.

We consider the residual graph G_f produced at the completion of the Ford-Fulkerson augmenting path algorithm.[1] Let the set S be all nodes reachable from s and T be $V \setminus S$. We wish to show that $C = (S, T)$ is a cut satisfying $|f| = c(S, T) = \sum_{(u,v)\in S\times T} c(u, v)$. This is true when the following is satisfied:

1. All edges $(u, v) \in S \times T$ are fully saturated by the flow. That is, $c_f(u, v) = 0$.
2. All reverse edges $(v, u) \in T \times S$ have zero flow. That is, $f(v, u) = 0$, or $c_f(v, u) = c(v, u)$.

The first condition is true by the way we constructed S and T, as if there existed a (u, v) where $c_f(u, v) > 0$, then v is accessible to s and ought to have been in S.

The second condition is true by the way the Ford-Fulkerson algorithm constructed reverse edges. If net flow was sent from v to u, then a reverse edge was constructed from u to v, so again, v is accessible to s, which is a contradiction.

Therefore, we have the flow

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

so we constructed a flow and a cut such that the flow $|f|$ is equal to the cut capacity $c(S, T)$, and we are done.

[1] If you're concerned that the Ford-Fulkerson algorithm will never terminate, there always exists a sequence of paths chosen such that it will. Edmonds-Karp is one example that always terminates.

9.2.3 Refinements of Ford-Fulkerson

As stated earlier, Ford-Fulkerson is limited by not specifying on which path to push flow. There are many algorithms that resolve this issue in different ways.

Edmonds-Karp

Edmonds-Karp fixes the problem by simply choosing the augmenting path of shortest unweighted length. This can be done easily using a BFS.

Algorithm 11 Edmonds-Karp

```
function ChoosePath(G_f(V, E_f), s, t ∈ V)                    ▷ BFS
    visited(v) denotes v has been added to queue
    prev(v) denotes vertex preceding v in BFS
    push s on queue Q
    visited(s) ← 1
    while Q is not empty do
        u ← top of Q
        for all neighbors v of u in G_f where visited(v) = 0 do
            push v on Q
            visited(v) ← 1
            prev(v) ← u
    pointer curr ← t
    while curr ≠ s do
        add curr to beginning of path p
        curr ← prev(curr)
    add s to beginning of p
    return p
function MaxFlow(G(V, E), s, t ∈ V)
    for all (u, v) ∈ V^2 do
        f(u, v) ← 0
        c_f(u, v) ← c(u, v)
    |f| ← 0
    while t can be reached from s do
        p ← ChoosePath(G_f, s, t)
        cap ← min_{i=1}^{m-1}(c_f(v_i, v_{i+1}))
        |f| ← |f| + cap
        AugmentPath(p)
    return |f|
```

The BFS is clearly $O(E)$. To complete our analysis, we must somehow bound the number of times we need to perform the BFS. To do this, we'll look at what pushing flow on a path does to our residual graph; in particular, how it affects our BFS traversal tree. Note that each vertex is on some level i in the BFS tree, characterized by the distance from the source s. For example, $L_0 = \{s\}$, L_1 contains all the neighbors of s, L_2 contains neighbors of neighbors not in L_0 or L_1, and so on.

We first claim that the level of any vertex in the graph is nondecreasing following an augment on a path p. If the augment saturates an edge, it may remove it from G_f, which cannot decrease the distance of any vertex from s. If the augment creates an edge $e = (u, v)$, that means we sent flow from v to u on the path p. Therefore, if v was originally level i, u must have been level $i + 1$. The level of u does not change by adding (u, v), and the level of v can either be i or $i + 2$, depending on whether edge (v, u) was deleted in the process. Either way, the level of all vertices is nondecreasing.

Now consider the bottleneck edge $e = (u, v)$ of an augmenting path p, where the level of u is i and the level of v is $i + 1$. The push operation deletes the edge e, but the level of v must stay at least $i + 1$. Now for the edge e to reappear in the graph G_f, flow must have been sent on the reverse edge $e' = (v, u)$ on some augmenting path p'. But on path p', u comes after v, which must be at least level $i + 1$. Therefore, u must be at least level i. But since the maximum level of a node that is connected to s is $V - 1$, an edge e can only be chosen as the bottleneck edge $\frac{V}{2}$ times, or $O(V)$.

There are E edges, each of which can be the bottleneck edge for $O(V)$ different augmenting paths, each of which takes $O(E)$ to process. Therefore, the Edmonds-Karp algorithm runs in $O(VE^2)$.

Dinic

Dinic's algorithm is another refinement of Ford-Fulkerson. Dinic's algorithm is similar to Edmonds-Karp in that we make use of the length of the shortest path from s to each vertex v. We first define the level graph G_L of G_f as the graph containing edges (u, v) only if $dist(v) = dist(u) + 1$.

We then define the blocking flow as a maximum flow f of a level graph G_L. Dinic's algorithm constructs a level graph G_L, augments the graph with a blocking flow, and repeats by finding another level graph. This process can happen $O(V)$ times, as with each blocking flow, $dist(t)$ increases.

Each BFS to compute the level graph is $O(E)$, and each blocking flow can be computed in $O(VE)$. Therefore, the run-time of Dinic's algorithm is $O(V^2E)$.

Using a link-cut tree, the computation of a blocking flow can be improved to $O(E \log V)$, improving the overall run-time to $O(VE \log V)$.

In networks with unit capacities, each blocking flow can be computed in $O(E)$, and the number of total blocking flows is $O(\min \sqrt{E}, V^{2/3})$, improving the overall run-time to $O(\min \sqrt{E}, V^{2/3}E)$.

9.2.4 Push-Relabel

Unfortunately, Dinic's algorithm is considerably complex, and even the much-improved bounds of the simpler $O(VE^2)$ Edmonds-Karp are admittedly bad. While the push-relabel method for solving the max flow problem does not have the fastest theoretical bounds, two of its implementations have complexities $O(V^3)$ and $O(V^2\sqrt{E})$ and are among the fastest in practice.

Generic Push-Relabel

Ford-Fulkerson and its variants all deal with global augmentations of paths from s to t. Push-relabel takes a different perspective, introducing the concept of a *preflow* and a *height* to make local optimizations that ultimately result in the maximum flow.

A preflow maintains the same properties as a flow but modifies the conservation of flow condition. Instead of total flow in equaling total flow out, flow in must be at least, and therefore can exceed, flow out. We denote the difference between flow in and flow out as the *excess* $e(v)$.

$$\begin{aligned} f(u,v) &\le c(u,v) && \forall (u,v) \in E \\ f(u,v) &= -f(v,u) && \forall (u,v) \in E \\ e(v) = \sum_{u \in V} f(u,v) &\ge 0 && \forall v \in V \setminus \{s\} \\ e(s) &= \infty. \end{aligned}$$

The definitions of the residual capacity $c_f(u,v)$, edge set E_f, and graph G_f are the same as they were defined before, except with a preflow f instead of a normal flow.

We call a vertex $v \in V \setminus \{s,t\}$ *active* if $e(v) > 0$. Therefore, a vertex besides the source or sink is active if more flows into the vertex than flows out. s and t are *never* active. A preflow with no active vertices is simply a flow, at which point the excess of the sink $e(t)$ represents the value $|f|$ of the flow.

We can *push* flow from a node u to a node v by moving as much of the excess $e(u)$ to v as the capacity of the edge $c_f(u,v)$ will allow.

function PUSH(edge (u,v))
  $\delta \leftarrow \min(e(u), c_f(u,v))$ ▷ $c_f(u,v) = c(u,v) - f(u,v)$
  $f(u,v) \leftarrow f(u,v) + \delta$
  $f(v,u) \leftarrow f(v,y) - \delta$
  $e(u) \leftarrow e(u) - \delta$
  $e(v) \leftarrow e(v) + \delta$

The idea of the push-relabel algorithm is to first push as much preflow as possible through local optimizations in the direction the sink. When a node can no longer push flow to the sink, it pushes the excess back towards the source to turn the preflow into a flow.

However, the difficulty here lies in establishing this sense of "direction" from the source to the sink. Remember that we simply push preflow along a single edge in the graph at a time, not along a whole path. Moving flow from the source to the sink along a path that goes from the source to the sink is easy; moving flow from the source to the sink through local pushes without the knowledge of the graph structure as a whole is indeed a much harder problem.

To resolve this issue, we introduce a *label* to each of the nodes. The label $h(u)$ represents the "height" of u. In real life, water flows from higher to lower ground. We want s to represent that high ground and t to represent the low ground. As we push preflow from s to t, vertices along the way represent height values between those of s and t. However, eventually we have to push preflow back to handle both excesses in flow and suboptimal previous pushes, à la Ford-Fulkerson, but this contradicts the concept of height as we can't flow both downhill and uphill. Therefore, we'll need to be able to *relabel* a node, changing the height $h(u)$ to something that allows preflow to flow back towards s. We will relabel h in a systematic way that allows us to direct the preflow through the graph.

For this labeling to be useful for us, we'll need to impose some more constraints that must be satisfied no matter how we change the graph G_f or the height function h.

$$h(u) \leq h(v) + 1 \forall (u,v) \in E_f$$
$$h(s) = |V|$$
$$h(t) = 0.$$

What does this mean? For our algorithm, we can push preflow along the edge from u to v only if $c_f(u,v) > 0$ and $h(u) > h(v)$, so $h(u) = h(v) + 1$. We call such an edge $(u,v) \in E_f$ *admissible*. Furthermore, for all vertices v that can reach t in E_f, $h(v)$ represents the lower bound for the length of any unweighted path from v to t in G_f, and for all vertices that cannot reach t, then $h(v) - |V|$ is a lower bound for the unweighted distance from s to v.

t will always represent the lowest node, so $h(t) = 0$ is a natural constraint. We'll first set the preflow values of all vertices v that can be immediately reached from s to $c(s,v)$, saturating all the out-edges of s. For any vertex v from which t can be reached, $h(v)$ represents the lower bound of the unweighted distance to t from v in the residual graph.

We want $h(s)$ to be a number large enough that will indicate that s has been disconnected to t, as we have already sent as much preflow possible from s in the direction of t by saturating all outgoing edges. Therefore, setting $h(s) = |V|$ is also natural. Since $h(s)$ represents the lower bound of the distance from s to t in G_f, and there are no paths from s to t in the residual graph, $|V|$ is a natural choice, since the longest possible path is $|V| - 1$.

Furthermore, we don't want any preflow sent back to s from a vertex v unless it is impossible to send any more preflow from v to t. If preflow is pushed from v to s, then $h(v) = |V| + 1$. If there existed a path v to t such that every edge is admissible, the path must have $|V| + 2$ vertices. This is true because for any two consecutive vertices v_i, v_{i+1} in the path, $h(v_i) = h(v_{i+1}) + 1$, but no path can have $|V| + 2$ distinct vertices.

This leads to the fact that the only nodes that can possibly continue to contribute to the final flow are active vertices v for which $h(v) < |V|$. A node with height at least $|V|$ does not have a valid path to t, and a node that is not active doesn't have any excess flow to push.

Now that I've explained the general idea behind the labeling constraints, it's time to actually describe what our relabeling process is. At first, the labels of all vertices besides the source start at 0. We only relabel a node v if it is active (therefore, it has excess flow it needs to push; $e(u) > 0$) but has no admissible out-edges in G_f (so it has no adjacent vertex on which it can push that excess flow). If a node has no admissible out-edges in G_f, every neighbor of u has a height label at least equal to $h(u)$. When we relabel a node, we always then *increase* the value of $h(u)$ to the least value where it can push flow onto another node.

function RELABEL(vertex u)
  $h(u) \leftarrow \min_{v|(u,v)\in E_f}(h(v)) + 1$ ▷ $(u,v) \in E_f \iff c_f(u,v) = c(u,v) - f(u,v) > 0$

Since we take the minimum height of all neighbors in the graph, we first try adjusting the height of u so that we can push flow from u to its neighbors that can possibly still reach t; that is, neighbors v satisfying $h(v) < |V|$. Once we try all these neighbors, we then increase the height of u to begin to push flow back towards s. We can always find such an edge, as any preflow pushed onto u must have also incremented the reverse edge from u back towards s.

Note that neither pushing on a admissible edge nor relabeling a vertex with no admissible out-edges changes the fact that h remains a valid labeling function.

The generic push-relabel algorithm simply pushes and relabels vertices until there are no active vertices and the preflow becomes a flow. This algorithm works because throughout the process, h remained a valid height function, and at the end, the preflow was converted into a flow. Since $h(s) = |V|$ and $h(t) = 0$, there is no augmenting path from s to t, so our flow is maximal.

Algorithm 12 Push-Relabel (Generic)

function PUSH(edge (u, v))
  if $e(u) > 0$ and $h(u) = h(v) + 1$ **then** ▷ push condition
    $\delta \leftarrow \min(e(u), c_f(u, v))$ ▷ $c_f(u, v) = c(u, v) - f(u, v)$
    $f(u, v) \leftarrow f(u, v) + \delta$
    $f(v, u) \leftarrow f(v, y) - \delta$
    $e(u) \leftarrow e(u) - \delta$
    $e(v) \leftarrow e(v) + \delta$
function RELABEL(vertex u)
  if $u \neq s, t$ and $e(u) > 0$ and $h(u) \leq h(v) \forall v | (u, v) \in E_f$ **then** ▷ relabel condition
    $h(u) \leftarrow \min_{v|(u,v)\in E_f}(h(v)) + 1$ ▷ $(u, v) \in E_f \iff c_f(u, v) = c(u, v) - f(u, v) > 0$
function MAXFLOW($G(V, E)$, $s, t \in V$)
  for all $v \in V \setminus \{s\}$ **do** ▷ initialize excess
    $e(v) \leftarrow 0$
  $e(s) \leftarrow \infty$
  for all $(u, v) \in V^2$ **do** ▷ initialize preflow
    $f(u, v) \leftarrow 0$
  for all neighbors $v \neq s$ of s **do** ▷ saturate edges to all neighbors of s
    $f(s, v) \leftarrow c(s, v)$
    $f(v, s) \leftarrow -c(s, v)$
    $e(v) \leftarrow c(s, v)$
  for all $v \in V \setminus \{s\}$ **do** ▷ preflow now has valid height function; initialize height
    $h(v) \leftarrow 0$
  $h(s) \leftarrow |V|$
  while we can still PUSH or RELABEL **do**
    PUSH or RELABEL
  return $e(t)$ ▷ $e(t) = |f|$

We can argue that this algorithm runs in $O(V^2E)$, which is already an improvement from Edmonds-Karp. However, just as Ford-Fulkerson could be sped up by specifying which augmenting paths to choose, we can do the same with the push-relabel algorithm, speeding it up by specifying a systematic method to choose an edge to push or a vertex to relabel.

Discharge

We first describe an auxiliary operation. For each vertex u, we'll need a way to visit in- and out-neighbors of u in a static cyclic order. This is easy with just a pointer; for vertex u, we'll call that pointer $curr(u)$. When the pointer passes through every element in the list of neighbors, we'll just reset it back to the first element.

```
function DISCHARGE(vertex u)
    while e(u) > 0 do                          ▷ perform an operation as long as u is active
        if curr(u) is at end of list of neighbors then
            RELABEL(u)
            reset curr(u)
        else
            if (u, curr(u)) is an admissible edge then
                PUSH((u, curr(u)))
            else
                move curr(u) to next neighbor of u
```

FIFO Selection

FIFO selection simply maintains a list of active vertices in a FIFO queue. We pop off the first vertex in the queue and discharge it, adding any newly-activated vertices to the end of the queue. This runs in $O(V^3)$.

Highest Label Selection

Highest label selection discharges the active vertex with the greatest height. This runs in $O(V^2\sqrt{E})$.

Improvements with Heuristics

Heuristics are meant to help relabel vertices in a smarter way. Bad relabelings are the slowest part of the algorithm, and improving the process can speed up max flow.

The *gap heuristic* takes advantage of "gaps" in the height function. Since a path of admissible edges consists of vertices whose heights decrease by exactly 1, the presence of a gap in height precludes the possibility of such a path. If there exists a value h' such that no vertex v exists such that $h(v) = h'$, then for every vertex v satisfying $h' < h(v) < |V|$, v has been disconnected from t, so we can immediately relabel $h(v) = |V| + 1$.

The *global relabeling heuristic* performs a backwards BFS from t every now and then to compute the heights of the vertices in the graph exactly.

Some dude on Codeforces[2] didn't have much luck improving performance with the global relabeling heuristic. I'd suggest sticking to the gap heuristic only.

9.2.5 Extensions

9.2.6 Bipartite Matchings

The bipartite matching problem is perhaps the most well-known problem solvable by flows.

[2] `http://codeforces.com/blog/entry/14378`