Network flows and shortest paths

Saad Mneimneh

1 Introduction

Let $G = (V, E)$ be a directed graph with two special vertices: a source s and a sink t. Every edge (u, v) has a capacity $cap(u, v) \geq 0$. For convenience, we define $cap(u, v) = 0$ if (u, v) in not an edge. More specifically, we work with a modified set of edges $\{(u, v) : (u, v) \in E \text{ or } (v, u) \in E\}$ with $cap(u, v) = 0$ if $(u, v) \notin E$. A flow is a real-valued function on edges with three properties:

- Skew symmetry: $f(u, v) = -f(v, u)$. If $f(u, v) > 0$ we say there is a flow from u to v.
- Capacity constraints: $f(u, v) \leq cap(u, v)$, if $f(u, v) = cap(u, v)$ we say that edge (u, v) is saturated.
- Flow conservation: $\sum_v f(u, v) = 0$, $\forall u \in V - \{s, t\}$.

The value $|f|$ of the flow is defined as $\sum_v f(s, v)$ and the problem is to find a flow f with maximum value.

2 A linear programming formulation

One way to tackle the maximum flow problem is by formulating it as a linear program and using the simplex algorithm. Here's a linear programming formulation of the problem.

$$\begin{array}{ll} \text{maximize} & \sum_v f(s, v) \\ \\ \text{subject to} & f(u, v) = -f(v, u) \\ & f(u, v) \leq cap(u, v) \\ & \sum_v f(u, v) = 0, \forall u \in V - \{s, t\} \end{array}$$

This linear program has a dual. We will explore this duality; however, we will address the problem from a graph theoretic perspective.

3 Flows and cuts, a duality

A important concept associated with a flow is that of a cut. A cut $X, V - X$ is a partition of the vertices into two sets X and $V - X$ such that $s \in X$ and $t \in V - X$. We define the capacity of a cut $cap(X, V - X) = \sum_{u \in X, v \in V - X} cap(u, v)$. A cut of minimum capacity is called a minimum cut. Similarly, we define the flow across a cut as $f(X, V - X) = \sum_{u \in X, v \in V - X} f(u, v)$. Here's an important observation:
Lemma:For any flow f and any cut $X, V - X$, $|f| = f(X, V - X)$.

Proof:

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

$$= \sum_{v} f(s, v) + \sum_{u \in X - \{s\}, v} f(u, v) + \sum_{u \in X, v \in X} f(u, v) = |f| + 0 + 0 = |f|$$

Since $X - \{s\}$ does not contains s or t, the value of the second term is 0 by flow conservation. The value of the third term is also 0, by skew symmetry.

Therefore, the value of the maximum flow is upper bounded by the value of the minimum cut. In fact, the max-flow min-cut theorem states that these two are equal (duality). To prove the theorem, we need to introduce the concepts of a residual graph and an augmenting path.

4 Residual graph, augmenting path, and the max-flow min-cut theorem

The residual graph R for a flow f is obtained from $G = (V, E)$ by changing the capacity (hence called residual capacity) of every edge (u, v) to $res(u, v) = cap(u, v) - f(u, v)$ (note that this is always ≥ 0). The set of edges of R is $\{(u, v) : res(u, v) > 0\}$.

An augmenting path for flow f is a path p from s to t in R. We can increase the value of f by increasing the flow on every edge of p by up to $\min_{(u,v) \in p} res(u, v)$. It turns out, this is a good mechanism for obtaining a maximum flow.

Theorem: (max-flow min-cut theorem). The following are equivalent:

1. f is a maximum flow
2. there is no augmenting path for f
3. $|f| = cap(X, V - X)$ for some cut $X, V - X$

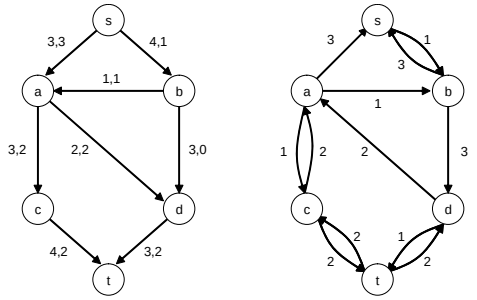

Figure 1: Residual graph: edges are labeled by their capacities and flows on the left, and their residual capacities (residual graph) on the right.

Proof: $1 \Rightarrow 2$: If there is an augmenting path p for f then we can increase the value of the flow by increasing the flow along p. $2 \Rightarrow 3$: Let X be the set of vertices reachable from s in R. Consider the cut $X, V - X$.

$$|f| = \sum_{u \in X, v \in V - X} f(u, v) = \sum_{u \in X, v \in V - X} cap(u, v) = cap(X, V - X)$$

where the second equality follows because $u \in X, v \in V - X$ implies that (u, v) is not an edge in R, i.e. $res(u, v) = 0$ which means $f(u, v) = cap(u, v)$. $3 \Rightarrow 1$: Since $|f| \leq cap(X, V - X)$, $|f| = cap(X, V - X)$ means that f is maximum flow and $X, V - X$ is a minimum cut.

Based on the above theorem, here's a basic algorithm for the maximum flow problem:

Ford and Fulkerson
for each (u, v)
    do $f(u, v) \leftarrow 0$
form R from G and f
while $\exists$ an augmenting path p (a path from s to t in R)
        do for each $(u, v) \in p$
                do $f(u, v) \leftarrow f(u, v) + \min_{(u,v) \in p} res(u, v)$
            form R from G and f

If all edge capacities are integers, the augmenting path algorithm increases the flow by at least one with every augmentation and, therefore, computes a maximum flow f^* in at most $|f^*|$ augmentations. Furthermore, $f^*(u, v)$ will be an integer for every edge (u, v). Unfortunately, if the capacities are large integers, the value of the maximum flow may be large, and the augmenting path algorithm may make many augmentations (see Figure 2 below).

Furthermore, if the capacities are irrational the algorithm may not halt, and although successive flow values converge they need not converge to the

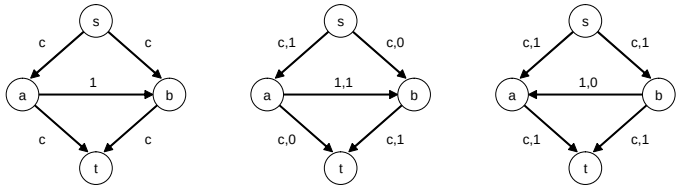

Figure 2: After $2c$ augmentations, alternately along $[s, a, b, t]$ and $[s, b, a, t]$, the flow is maximum.

value of the maximum flow. Thus if the algorithm is to be efficient we must select augmenting paths carefully. We discuss two possibilities in the following sections.

5 Maximum capacity augmentation

A natural way to select augmenting paths is to always augment along a path of maximum residual capacity, i.e. a path p in the residual graph that maximizes $\min_{(u,v)\in p} res(u, v)$. The basis for this method is the following observation:

Lemma: Starting from the zero flow, there is a way to construct a maximum flow in at most m augmentations.

Proof: Given the maximum flow f^*, consider the graph G^* obtained from G by removing edges (u, v) with $f^*(u, v) \leq 0$. Repeatedly, find a path p from s to t in G^*, and decrease the flow f^* along p by $\min_{(u,v)\in p} f * (u, v)$. At least one edge on p will now have zero flow; remove all such edges. Each path finding step deletes at least one edge of G^*; thus, this algorithm halts in at most m steps, having reduced f^* to a flow of value zero (although there may still be cycles of flow). Constructing the maximum flow in at most m augmentations corresponds to the backward process.

Based on the above lemma, one can show that maximum capacity augmentation is efficient when capacities are integers. In fact, we have the following theorem:

Theorem: Maximum capacity augmentation produces successive flow values that converge to the value of the maximum flow. If the capacities are integers, the algorithm finds a maximum flow in $O(m \log c)$ augmenting steps, where c is the largest edge capacity.

Proof: We will prove the second part of the theorem (integer capacities). Let f be any flow and f^* be a maximum flow. Consider the residual graph R of f. Starting from the zero flow on R, there are at most m augmenting paths whose residual capacities sum to $|f^*| - |f|$. Therefore, the maximum capacity

augmenting path has residual capacity at least $(|f^*| - |f|)/m$. Now consider a sequence of $2m$ maximum capacity augmentations, starting from flow f. At least of these must augment the flow by an amount $\leq (|f^*| - |f|)/(2m)$. Therefore, after $2m$ or fewer maximum capacity augmentations, the capacity of a maximum capacity augmenting path is reduced by a factor of two. Since this capacity is initially at most c and is at least one unless the flow is maximum, after $O(m \log c)$ maximum capacity augmentations the flow must be maximum.

It remains to show a method for finding a maximum capacity augmenting path, and this will determine the total running time of the algorithm. We can use a suitable modification of Dijkstra's algorithm for finding shortest paths. Dijkstra's algorithm works by first setting $d[s] = 0$ and $d[u] = \infty$ for all vertices $u \neq s$, then repeatedly selecting the vertex u with the smallest distance $d[u]$, and relaxing its edges, i.e. setting $d[v] = \min(d[v], d[u] + w(u, v))$, where v is a neighbor of u and $w(u, v)$ is the weight of the edge (u, v). In fact, relaxing edges enough times guarantees a correct computation of shortest distances from s as long as we have no negative weight cycles (Bellman-Ford algorithm), but Dijkstra's algorithm has the advantage of relaxing every edge only once. Note, however, that Dijkstra's algorithm works only when weights are non-negative. In the following pseudocode, we assume that all vertices have been properly initialized to unvisited, $d[u]$ to ∞, and $d[s]$ to 0. The actual shortest path tree can be obtained by updating $\pi[v]$, the parent of v, to u each time $d[v]$ changes to $d[u] + w(u, v)$.

$\underline{Dijkstra(G)}$
$Q \leftarrow \{s\}$
while Q is not empty
  $u \leftarrow \arg\min_{u \in Q} d[u]$
  $Q \leftarrow Q - \{u\}$
  $visited[u] \leftarrow$ true
  for each $v \in adj[u]$
    do $d[v] \leftarrow \min(d[v], d[u] + w(u, v))$
      if not $visited[v]$
        then $Q \leftarrow Q \cup \{v\}$

The correctness of Dijkstra's algorithm can be derived by induction. Our invariant is the following: when a vertex u is removed from Q, $d[u]$ is the shortest distance from s to u, denoted by $d^*[u]$. Let u be the first vertex to violate this property and assume $d[u] > d^*[u]$ when u is about to be removed from Q. Consider a shortest path p from s to u. Since $s \notin Q$ and $u \in Q$, there must be $x \notin Q$ and $y \in Q$ such that (x, y) is an edge of the path. Note that the path from s to x along p is shortest, and the path from s to y along p is shortest. Therefore, $d[y] = d^*[y]$ (because $d[y] = d[x] + w(x, y)$ has been considered). Now $d^*[u] = d^*[y] + d^*[y \rightarrow u]$. Since all weights are non-negative, $d^*[u] \geq d^*[y] = d[y]$. Therefore, $d[u] > d[y]$ contradicting the fact that u will be removed from Q.

Note that Dijkstra's algorithm finds for each vertex u a path p such that $p = \arg\min_p \sum_{(u,v)\in p} w(u,v)$. What we need, however, is a path $p = \arg\max_p \min_{(u,v)\in p} res(u,v)$. Therefore, by considering weights to be the negatives of the capacities, we seek a path $p = \arg\min_p \max_{(u,v)\in p} w(u,v)$, where $w(u,v) = -res(u,v)$. With this modification, the shortest path is now a path that minimizes not the distance (total weight of edges on the path), but the largest weight on the path. Having negative weights is not a problem for this modified Dijkstra's algorithm, as we will see shortly.

$\underline{Modified - Dijkstra(G)}$
$Q \leftarrow \{s\}$
while Q is not empty
  $u \leftarrow \arg\min_{u\in Q} d[u]$
  $Q \leftarrow Q - \{u\}$
  $visited[u] \leftarrow$ true
  for each $v \in adj[u]$
    do $d[v] \leftarrow \min(d[v], \max(d[u], w(u,v)))$
      if not $visited[v]$
        then $Q \leftarrow Q \cup \{v\}$

The correctness of the modified Dijkstra's algorithm can be derived also by induction. Our invariant is the same: when a vertex u is removed from Q, $d[u] = d^*[u]$, where *shortest* path is now a path that minimizes the maximum weight along the path, and $d^*[u]$ is that weight. Let u be the first vertex to violate this property and assume $d[u] > d^*[u]$ when u is about to be removed from Q. Consider a *shortest* path p from s to u. Since $s \notin Q$ and $u \in Q$, there must be $x \notin Q$ and $y \in Q$ such that (x,y) is an edge of the path. The path p can be chosen so that the path from s to x along p is *shortest*, and the path from s to y along p is *shortest*. Therefore, $d[y] = d^*[y]$ (because $d[y] = max(d[x], w(x,y))$ has been considered). Now $d^*[u] = \max(d^*[y], \max y \rightarrow u)$, thus $d^*[u] \geq d^*[y] = d[y]$ (regardless of the weights). Therefore, $d[u] > d[y]$ contradicting the fact that u will be removed from Q.

With proper implementation, Dijkstra's algorithm runs in $O(m \log n)$ time using a heap for Q. Therefore, the total running time for finding a maximum flow based on the maximum capacity augmentation algorithm is $O(m^2 \log n \log c)$.

It is worth noting that a special case of Dijkstra's algorithm when all weights are equal (e.g. to 1) reduces to a breadth first search on the graph as illustrated below. The algorithm runs in $O(m)$ time starting at vertex s and explores all vertices (and edges) reachable from s. The $O(m)$ bound for the running time is possible due to the simplified operations on Q (every newly added vertex to Q has the largest distance so far). Again, the BFS tree can be obtained by updating $\pi[v]$, the parent of v, to u each time $d[v]$ changes to $d[u] + 1$. We will use BFS in the following section.

```
BFS(G)
Q ← {s}
while Q is not empty
    u ← first element of Q
    Q ← Q − {u}
    visited[u] ← true
    for each v ∈ adj[u]
        do d[v] ← min(d[v], d[u] + 1)
            if not visited[v] and v ∉ Q
              then add v to the end of Q
```

6 Shortest path augmentation

Another way to select augmenting paths is along shortest paths, where we measure the length of the path as the number of edges on the path, i.e. all weights are 1. If we always choose a shortest augmenting path, we can show that we will have at most $O(mn)$ augmentations, regardless of whether edge capacities are integers or not. Since a shortest path can be found by BFS, which runs in $O(m)$, the total running time for finding a maximum flow based on the shortest path augmentation algorithm is $O(m^2n)$. However, we develop here a better way in which we augment along all shortest paths simultaneously. For this, we need the concept of a blocking flow.

A flow is blocking if every path from s to t contains a saturated edge (recall a saturated edge (u, v) is such that $f(u, v) = cap(u, v)$. Given a flow f, let $d[u]$ be the distance from s as computed by the BFS algorithm on the residual graph R of flow f. Define the level graph L containing only vertices reachable from s in R and edges (u, v) of R such that $d[v] = d[u] + 1$. Thus, L contains every shortest augmenting path and is constructed in $O(m)$ time by BFS. Our augmentation works, not by simply augmenting along a shortest path in R, but by finding a blocking flow for L. We will call such an augmentation a blocking step.

Theorem: There are at most $n - 1$ blocking steps.

Proof: Let f be the current flow, R its residual graph, L its level graph, R' the residual graph after a blocking step, and $d[u]$ and $d'[u]$ the distances computed by BFS on R and R' respectively. Each edge $(u, v) \in R$ has $d[v] \leq d[u] + 1$. Each edge in R' is either an edge in R or the reverse of an edge in L. Thus each edge $(u, v) \in R'$ also has $d[v] \leq d[u] + 1$. This means that $d'[u] \geq d[u]$ and, in particular, $d'[t] \geq d[t]$. We will show that $d'[t]$ cannot be equal to $d[t]$. Assume otherwise and let p be a shortest path from s to t in R'. This means that $d'[v] = d'[u] + 1$ for every edge (u, v) on p, which means that $d[v] = d[u] + 1$ for every edge (u, v) on p (because $d'[u] \geq d[u]$), which means every such edge is also in L. This contradicts the fact that at least one such edge is saturated by the blocking step and does not appear in R'. Therefore, $d'[t] > d[t]$. Since the

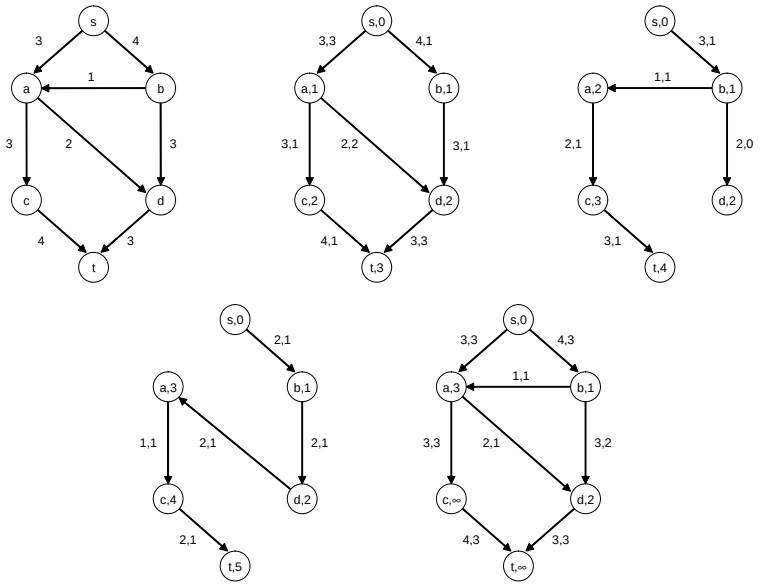

Figure 3: Initial graph followed by three successive level graphs with nodes labeled by their distances, and edges by their residual capacity and flow. The final graph shows the final flow.

shortest distance from s to t is at least one, at most $n-1$, and increases by at least one or becomes undefined with each blocking step, the number of blocking steps is at most $n-1$.

It remains to show a method for finding a blocking flow, and this will determine the total running time of the algorithm. We will use a modification of DFS as follows:

$\underline{Blocking - flow(L, s, t)}$
while s has outgoing edges
run DFS-VISIT(s) on L with the following two modifications

> case 1: If t is reached, augment along the path p consisting of the edges in the recursion stack, by $\min_{(u,v)\in p} res(u,v)$. Update capacities and delete all saturated edges on that path. Stop DFS-VISIT.
>
> case 2: If a vertex v with no outgoing edges is reached, delete the edge (u,v) at the top of the recursion stack. Continue DFS-VISIT.

The above algorithm deletes an edge (u, v) from L only if (u, v) is saturated or every path from v to t contains a saturated edge. It follows that the algorithm constructs a blocking flow.

To analyze the running time of this algorithm, think about the recursion stack of edges. If an edge is pushed onto the stack, it will either lead to an augmenting path (case 1), or it will be popped off the stack (case 2). In case 1, we end up with $O(n)$ edges in the stack (length of a path from s to t) and this requires $O(n)$ time for the augmentation, which will then delete at least one edge. Therefore, the total contribution of case 1 is $O(nm)$. When an edge is popped off the stack, it is deleted and, therefore, every such step must be associated with the deletion of an edge. Thus, the total contribution of case 2 is $O(m)$. The total running time is $O(nm) + O(m) = O(nm)$.

Putting it all together, we have an algorithm for finding a maximum flow that runs in $O(n^2m)$ time. Another algorithm, based on the concept of a pre-flow with $\sum_v f(u, v) \geq 0$ (flow conversation is not satisfied), which is then converted into a flow, computes a blocking flow in $O(n^2)$ time, resulting in an $O(n^3)$ running time for maximum flow. Sleator and Tarjan discovered a way to compute a blocking flow in $O(m \log n)$ using a special data structure that involves linking and cutting trees. This results in an $O(nm \log n)$ running time for maximum flow, which is better than $O(n^3)$ for sparse graphs (when $m = o(n^2)$).