15-451/651: Design & Analysis of Algorithms Feb 28, 2023
Lecture #12: Network Flows II last changed: February 28, 2023

The Ford-Fulkerson algorithm discussed in the last class takes time $O(mF)$, where F is the value of the maximum flow, when all capacities are integral. This is fine if all edge capacities are small, but if they are large numbers then this could even be exponentially large in the description size of the problem. In this lecture we examine some improvements to the Ford-Fulkerson algorithm that produce much better (polynomial) running times regardless of the value of the max flow or capacities.

Objectives of this lecture

In this lecture, we will:

- See an algorithm for max flow with polynomial running time (Shortest Augmenting Paths)
- Analyze an even better algorithm for max flow that runs in polynomial time (Dinic's)

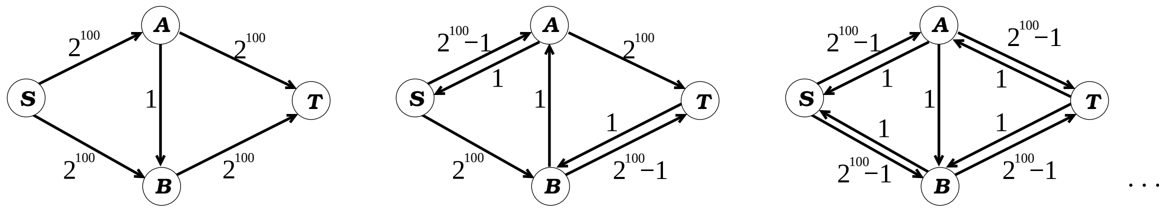
Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 26, Maximum Flow
- DPV, *Algorithms*, Chapter 7.2, Flows in Networks

1 Network flow recap

Recall that in the maximum flow proble, we are given a directed graph G, a source s, and a sink t. Each edge (u, v) has some capacity $c(u, v)$, and our goal is to find the maximum flow possible from s to t. Last time we looked at the Ford-Fulkerson algorithm, which we used to prove the min-cut max-flow theorem, as well as the integrality theorem for flows. The Ford-Fulkerson algorithm is a greedy algorithm: we find a path from s to t of positive capacity and we push as much flow as we can on it (saturating at least one edge on the path). We then describe the capacities left over in a "residual graph", which accounts for remaining capacity as well as the ability to redirect existing flow (and hence "undo" bad previous decisions) and repeat the process, continuing until there are no more paths of positive residual capacity left between s and t. We then proved that this in fact finds the maximum flow.

Assuming capacities are integers, the basic Ford-Fulkerson algorithm could make up to F iterations, where F is the value of the maximum flow. Each iteration takes $O(m)$ time to find a path using DFS or BFS and to compute the residual graph. (We assume that every vertex in the graph is reachable from s, so $m \geq n - 1$.) So, the overall total time is $O(mF)$. This is fine if F is small, like in the case of bipartite matching (where $F \leq n$). However, it's not good if capacities are large and F could be very large. Here's an example that could make the algorithm take a very long time. If the algorithm selects the augmenting paths $s \to A \to B \to t$, then $s \to B \to A \to t$, repeating..., then each iteration only adds one unit of flow, but the max flow is 2^{101}, so the algorithm will take 2^{101} iterations. If the algorithm selected the augmenting paths $s \to A \to t$ then $s \to B \to t$, it would be complete in just two iterations! So the question on our minds today is can we find an algorithm that provably requires only polynomially many iterations?



2 Shortest Augmenting Paths Algorithm (Edmonds-Karp)

There are several strategies for selecting better augmenting paths than arbitrary ones. Here's one that is quite simple and has a provable polynomial runtime. Its called the Shortest Augmenting Paths algorithm, or the Edmonds-Karp algorithm. The name of the algorithm might give away a slight hint of what it does.

Algorithm: Shortest Augmenting Paths (Edmonds-Karp)

Edmonds-Karp implements Ford-Fulkerson by selecting the *shortest* augmenting path each iteration.

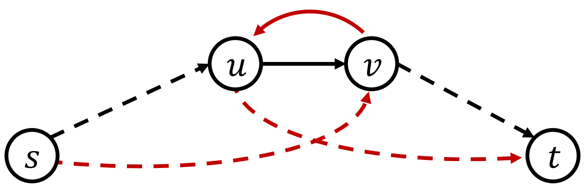
Unsurprisingly, the Shortest Augmenting Paths (Edmonds-Karp) algorithm works by always picking the *shortest* path in the residual graph (the one with the fewest number of edges). By example, we can see that this would in fact find the max flow in the graph above in just two iterations, but what can we say in general? In fact, the claim is that by picking the shortest paths, the algorithm makes at most mn iterations. So, the running time is $O(nm^2)$ since we can use BFS in each iteration. The proof is pretty neat too.

Theorem: Runtime of Edmonds-Karp

The Shortest Augmenting Paths algorithm (Edmonds-Karp) makes at most mn iterations.

Proof. Let d be the distance from s to t in the current residual graph. We'll prove the result by showing:

Claim (a): d never decreases Let's consider one iteration of the algorithm. Before adding flow to the augmenting path, every vertex v in G has some distance d_v from the source vertex s *in the residual graph*. Suppose the augmenting path consists of the vertices $v_1, v_2, \ldots v_k$. What can we say about the distances of the vertices? Since the path is by definition a *shortest path*, it must be true that $d_{v_i} = d_{v_{i-1}} + 1$, that is, every vertex is one further from s. Now perform the augmentation and consider what changes in the residual graph. Some of the edges (at least one) become *saturated*, which means that the flow on the edge reaches its capacity. When this happens, that edge will be removed from the residual graph. But another edge might appear in the residual graph! Specifically, when $e = (u, v)$ is saturated, $e' = (v, u)$ may appear in the residual graph as a back edge (if it doesn't exist already). Can this lower the distance of any vertex? No, because $d_v = d_u + 1$, so adding an edge from v to u can't make a shorter path from s to t.



Therefore, since the distance to any vertex can not decrease, d can not decrease.

Claim (b): every m iterations, d has to increase by at least 1 Each iteration saturates (fills to capacity) at least one edge. Once an edge is saturated it can not be used because it will not appear in the residual graph. For the edge to become usable again, it must be the case that its back edge in the residual graph is used, which means that the back edge needs to appear on the shortest path. However, if $d_v = d_u + 1$, then it is not possible for the back edge (v, u) to be on a shortest path, so this can *only* occur if d increases. Since there are m edges, d must increase by at least one every m iterations.

Since the distance between s and t can increase at most n times, in total we have at most nm iterations. $\square$

This shows that the running time of this algorithm is $O(nm^2)$. Note that this is true for **any** capacities, including large ones and non-integer ones. So we really have a polynomial-time algorithm for maximum flow!

3 Dinic's Algorithm

The Edmonds-Karp algorithm already gives us polynomial runtime, but now we will try to do better. The key idea is going to be to eliminate some redundancy in the Edmonds-Karp algorithm. Here are two observations that can guide us towards a better algorithm. Let d be the distance from s to t in the current residual graph:

- Our analysis above implies that there are up to mn iterations, but d can only increase n times, so there could be up to m iterations for each value of d
- The breadth-first search algorithm, which we use to find a shortest path, doesn't only return a single shortest path. It finds *all of the distances* in the graph from s, which means it encodes *every possible shortest path* of length d.

So maybe we don't actually need to do a BFS for every augmenting path. Could we do a BFS and use the resulting information to find *multiple shortest augmenting paths* such that there are none left? That is exactly the idea of *Dinic's algorithm*. The concept turns out to be extremely useful in the theory of network flow, so it even has a name. Its called a *blocking flow*.

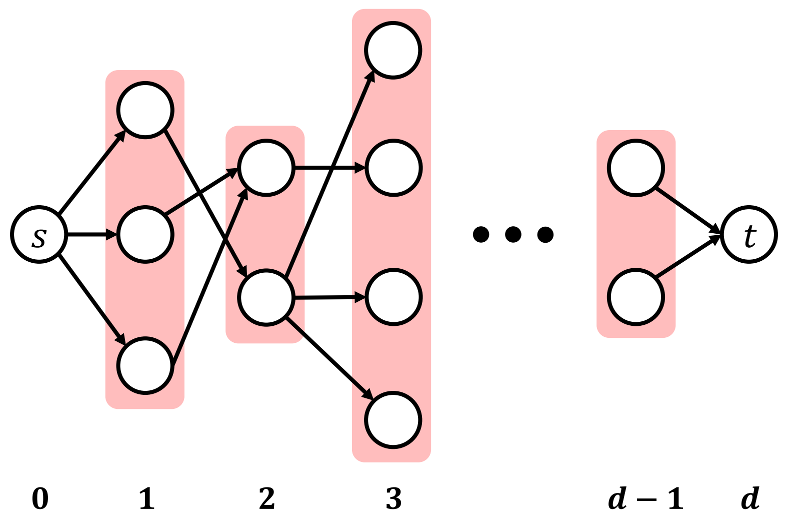
Definition: Blocking Flow

A *blocking flow* in a residual graph G_f is a union of shortest augmenting paths in G_f such that every shortest path in G_f has at least one edge that gets saturated.

Another way to think of it is that a blocking flow is a collection of augmenting paths of length d such that after augmenting all of them, there are no more paths in the residual graph of length d, which is exactly what we want. It completely "uses up" distance d. If we find a blocking flow and augment along it, then d must increase, and hence we can only find n blocking flows before we are done. Our goal is therefore reduced to designing a fast algorithm for finding blocking flows!

3.1 Finding blocking flows: the layered graph

Our tool for finding blocking flows is going to be a way of visualizing all of the shortest paths in the graph, called the *layered graph*. Suppose we run a BFS from s to compute the shortest path distance d_v for every vertex $v \in V$. By definition, $d = d_t$. Now we visualize the graph in the following way: lay out all of the vertices of distance one in a "layer", followed by all vertices of distance two, and so on. The following diagram illustrates the idea.



The layered graph L only includes the edges that go from one distance layer to the next higher layer. Note that there can not exist any edges that go from a layer to a layer more than one higher (because then the distance to that vertex would actually be lower than our layered graph implies). There can be edges in the original graph that go from a vertex to a vertex in a lower layer, but we ignore those because they can not be part of a shortest path.

3.2 An algorithm for blocking flows

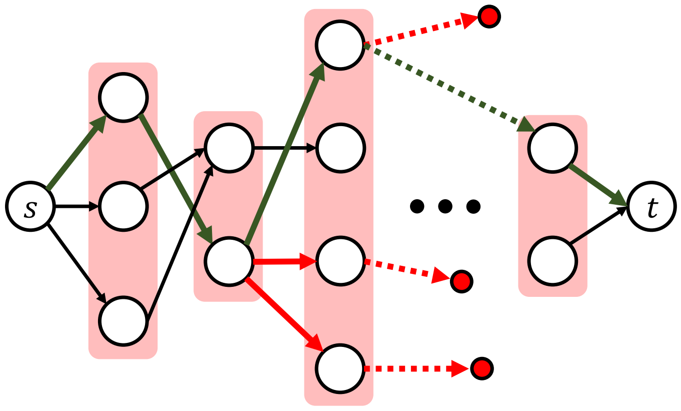
We can compute the layered graph implicitly with a BFS from s in $O(n + m)$ time. Although it doesn't matter in theory, in practice, it is a nice optimization to realize that you don't actually have to make an explicit copy of the layered graph, rather, you can just read it out of the original graph by ignoring all edges (u, v) that **don't** satisfy $d_v = d_u + 1$. In other words, the layered graph is just the original graph but only counting edges that are actually on a shortest path from s.

Key Idea: Implicit layered graph

The layered graph is the subgraph of G_f containing only edges (u, v) that satisfy $d_v = d_u + 1$.

Once we have the layered graph, at a high level, we are just going to find augmenting paths in it using DFS. How fast would this be with a naive implementation? Well, each DFS could cost $O(m)$ and there could be up to m augmenting paths in a blocking flow since each of them saturates one edge, so this could take $O(m^2)$ time. This so far isn't an improvement over Edmonds-Karp since the whole algorithm would take $O(nm^2)$ time to find up to n blocking flows. Somehow we need to speed this up and make finding the augmenting paths in the blocking flow more efficient.

Lets consider what actually happens during a DFS that is searching for an augmenting path in the layered graph. The search has to follow two rules: It can only use edges in the layered graph (i.e., $d_v = d_u + 1$) and the edge must not already be saturated, so $f(u, v) < c(u, v)$. Here's a key idea. A successful augmenting path will always contain at most $n - 1$ edges, otherwise there would be repeat vertices which is pointless. But the naive DFS could take $O(n + m)$ time, why? Because it might visit a bunch of edges that end up not actually being part of the final augmenting path, because the search reached a dead end.



Here's a diagram illustrating the idea. Suppose the green path is the successful augmenting path found. The red edges/paths represent unsuccessful branches of the DFS that did not manage to find an unsaturated path to t. If the DFS could magically only take the correct branches every time, it would take $O(n)$ and we would be very happy! So here's the idea. Any time our DFS finds an unsuccessful branch, like the ones marked in red above, we know that future DFS iterations shouldn't bother to try those paths, because they didn't work last time and they won't work this time either! So, once an edge has been searched and found to be unsuccessful, we can mark it as "dead" and never search it in future iterations.

3.3 Runtime analysis

Given the strategy above, we claim that we can find a blocking flow faster than the naive strategy.

Theorem: Runtime of blocking flows

A blocking flow can be found in $O(nm)$ time.

Proof. First, we do a BFS to find all of the distances to each vertex and establish the layered graph. This takes $O(n+m)$ time. Now, we do up to m DFS's to find augmenting paths in the layered graph. Naively, these take $O(m)$ time each and we are in trouble. Instead, we use the strategy above of marking edges on unsuccessful search paths as "dead" and never search them in a future DFS. Each edge in the graph can only be searched unsuccessfully once across all DFS iterations, and the running time of each DFS is therefore

$$n + \text{number of new edges marked dead}$$

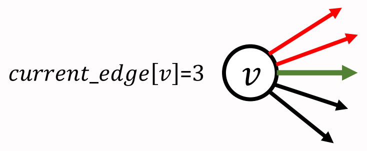
Summing this up over m DFS's, the total running time is

$$\begin{aligned}
&\sum_{i=0}^{m} \left(n + \text{number of new edges marked dead in iteration } i\right), \\
&= nm + \sum_{i=0}^{m} \left(\text{number of new edges marked dead in iteration } i\right), \\
&= nm + \text{total number of new edges ever marked dead}, \\
&= nm + m, \\
&= O(nm).
\end{aligned}$$

What we have basically done here is an aggregate amortized analysis to show that the amortized cost of each DFS in the blocking flow is at most $O(n)$. Therefore, we can find a blocking flow in $O(nm)$ time. □

Remark: Implementing the blocking flow algorithm

How should we actually implement the blocking flow algorithm in practice? Its not obvious how we actually mark edges as "dead" during a search. We could copy the entire graph and then delete the edges that are marked dead from the adjacency list, but this is quite inefficient since it requires (a) copying the entire graph and (b) deleting edges from the adjacency list is not efficient if it is stored as an array-based list, so we'd have to use a linked list, BST, or dynamic hash table. Here's a common alternative that can be used to get a fast and practical implementation. For each vertex, store a counter that marks the index of the current edge that we are searching from.



When the search arrives at vertex v, search from *current_edge*$[v]$ and if successful, do not modify *current_edge*$[v]$ (it may still be useful to use the same edge again in the next augmenting path if it is not yet saturated). If the edge is saturated, or the search is unsuccessful and backtracks, then increment *current_edge*$[v]$, which marks the edge as dead, and continue searching from the next edge.

By using this algorithm for blocking flows, we get an algorithm for max flow in $O(n^2m)$ time, which is an improvement over Edmonds-Karp's $O(nm^2)$.

3.4 Dinic's algorithm for unit-capacity graphs

A remarkable thing about Dinic's algorithm is that it seems to perform really well in practice – much better than the $O(n^2m)$ bound would suggest. In fact, we can prove that for many classes of common real-world graphs, it is actually asymptotically faster!

Lemma 1: Finding a blocking flow in a unit-capacity graph

If every edge in the graph has capacity one, then a blocking flow can be found in $O(m)$ time.

Proof. First note that the unit capacity property is preserved in all residual graphs. We now just make a slight modification to the analysis above. Previously, we said that there can be up to $O(m)$ augmenting paths, and each of them is at most $n-1$ edges long, so the total runtime of finding a blocking flow was $O(nm)$ (the total length of all the augmenting paths), plus the cost of visiting the dead-end edges at most once which was $O(m)$. We can be tighter when the edges are all unit capacity.

Since the edges have capacity one, no edge can be used in multiple augmenting paths (they will all be edge disjoint now), so the total length of all augmenting paths is at most m. The cost of finding the blocking flow is the sum of the lengths of the augmenting paths, which is m, plus the cost of visiting the dead-end edges, which is an additional $O(m)$, so the whole blocking flow takes $O(m)$. $\square$

Lemma **??** shows that in unit-capacity graphs, we can therefore find the max flow in at most $O(mn)$ time since we need at most n blocking flows. It turns out that we're not done and we can still improve more!

Lemma 2: Number of blocking flows in a unit-capacity graph

If every edge in the graph has capacity one, then we need at most $2\sqrt{m}$ blocking flows.

Proof. Let us consider the state of the residual network after k blocking flows have been found, for any arbitrary value of k. At this point, d must be at least k since each blocking flow increases the distance. Now we ask, what is the max flow of this residual graph, i.e., how much more flow can we augment in the graph before we hit the max flow? Well, since the distance is currently at least k, every augmenting path has length at least k, and there are m total edges in the graph. Since the edges are unit capacity, each can only be used in one path, so we can form at most m/k augmenting paths. Each augmenting path has capacity one, and hence the max flow in the residual network is at most m/k.

Since each blocking flow adds at least one unit of flow, this means that we need at most m/k additional blocking flows. Since we have already done k blocking flows, the total number of blocking flows across the whole algorithm is at most

$$k + \frac{m}{k}.$$

Now what was k again? Well it could be any arbitrary constant we like, so if we can pick k to minimize this expression, we get a good bound on the number of blocking flows needed. We could do calculus to find the minimum, but that's too hard, so here's an easier trick that is quite a nice one to know. We want to minimize the sum of a term that is increasing with k and a term that is decreasing with k. Asymptotically (within a factor of two), we can minimize such an expression by finding when they are equal[1]. So when $k = m/k$, we have $k = \sqrt{m}$, and therefore number of blocking flows is at most $2\sqrt{m}$. $\square$

This shows that on a unit-capacity network, Dinic's algorithm runs in time $O(m\sqrt{m})$. For sparse graphs ($m = \Theta(n)$), this is an improvement over $O(nm)$. For dense graphs ($m = \Theta(n^2)$), it is about the same.

[1]This trick works because $f(k) + g(k) \leq 2\max(f(k), g(k))$, and the maximum is minimized when the two are equal.