CMSC 451: Lecture 13
Network Flow Algorithms

Algorithmic Aspects of Network Flow: In the previous lecture, we introduced the network-flow problem. We discussed concepts like the residual network, augmenting paths, the Ford-Fulkerson algorithm, cuts, and the Max-Flow/Min-Cut Theorem. We used the Max-Flow/Min-Cut Theorem to prove that the Ford-Fulkerson algorithm is correct. Before reading this lecture, please refer back to that one for these definitions. In this lecture we discuss the running time of the Ford-Fulkerson algorithm, and introduce some more efficient alternatives.

Analysis of Ford-Fulkerson: Before discussing the worst-case running time of the Ford-Fulkerson algorithm, let us first consider whether it is guaranteed to terminate. We assume that all edge capacities are integers.[1] Every augmentation by Ford-Fulkerson increases the flow by a strictly positive integer amount. Since the maximum flow is finite, Ford-Fulkerson must terminate after a finite number of iterations. Thus, we have the following.

Lemma: Given an s-t network with integer capacities, the Ford-Fulkerson algorithm terminates. Furthermore, it produces an integer-valued flow function.

Recall our convention that $n = |V|$ and $m = |E|$. Since we assume that every vertex is reachable from s, it follows that $m \geq n - 1$. Therefore, $n = O(m)$. Running times of the form $O(n + m)$ can be expressed more simply as $O(m)$.

As we saw last time, the residual network can be computed in $O(n + m) = O(m)$ time and an augmenting path can also be found in $O(m)$ time. Therefore, the running time of each augmentation step is $O(m)$. How many augmentations are needed? Unfortunately, the number could be very large. To see this, consider the example shown in Fig. 1.

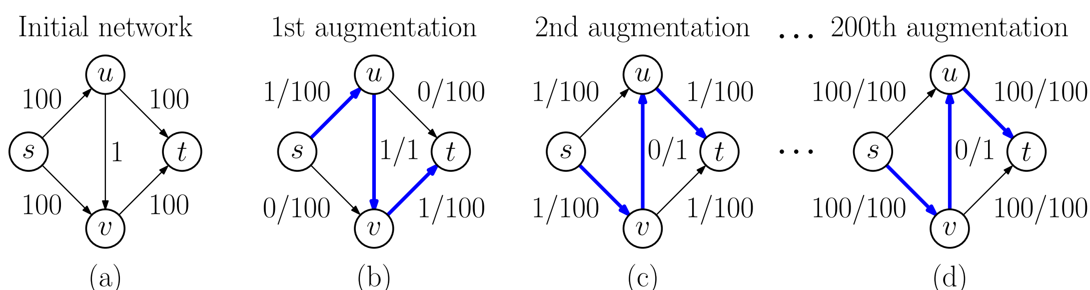

Fig. 1: Bad example for Ford-Fulkerson.

Suppose that we (foolishly) elect to augment using a path from s to t that uses the vertical edge in the middle, alternately increasing its flow to 1 and reducing it to 0. It would take 200 augmentation steps to converge. We could replace 100 with whatever huge value we want and make the running time arbitrarily high.

If we let $|f|$ denote the final maximum flow value, we can see from the example that the number of augmentation steps can be as high as $|f|$. If we make the reasonable assumption

[1] This is reasonable, since we can always apply a uniform scale factor to convert fractional values to integers.

that each augmentation step takes at least $\Omega(m)$ time, the total running time can be as high as $\Omega(m|f|)$. In general, if C is any upper bound on the maximum flow, the running time of the Ford-Fulkerson algorithm is $O(mC)$.

Scaling Algorithm: In the above example (Fig. 1), we made the apparently foolish decision to augment on a path of very *low capacity*. Intuitively, it would be smarter to select the *maximum capacity* augmenting path. We can compute the maximum capacity s-t path by a variant of Dijkstra's algorithm in $O(m \log n)$ time. However, we don't need to exact maximum. It suffices to pick any path whose capacity is "close" to maximum. Such a path can be computed in just $O(m)$ time. This is the essence of the *scaling algorithm*. This algorithm (and generally the idea of solving optimization problems through scaling) was introduced by Hal Gabow in the mid 1980's.

The idea of scaling is to eliminate (actually, just disregard) all the edges of small capacity, focusing instead on the high-capacity edges. Once we have pushed as much flow as we can through these "heavy hitters," we can consider edges of smaller and smaller capacities. Eventually, we will get down to the smallest capacity edges.

Let's assume that all of the capacities are integers. (If not, scale them up uniformly so they are.) To get this process started, we'll need to define the highest capacity edges. To do this, let's start by computing a crude upper bound on the maximum flow value. We can do this by taking the capacity on any valid cut.[2] For simplicity, lets take the cut that has s on one side and all the other vertices on the other. Let C denote the capacity of this cut, that is,

$$C = \sum_{(s,v) \in E} c(s, v).$$

The algorithm makes use a *scaling* (or *resolution*) *parameter*, denoted Δ. The value of Δ will be a power of 2. Initially, its value will be close to C, and we will progressively decrease Δ until it equals 1. Intuitively, we will ignore all edges whose capacities are smaller than Δ. This implies that the early phases of the algorithm will focus on pushing flow through the fattest pipes and the final stages will work with the skinniest pipes.

Initially, let's Δ to be the largest power of 2, such that $\Delta \leq C$. (Formally, $\Delta \leftarrow 2^{\lfloor \lg C \rfloor}$.) Given any flow f, define $G_f(\Delta)$ to be the residual network consisting *only of edges of residual capacity at least* Δ. An example is shown in Fig. 2.

Recall that an *augmenting path* in a residual network is an s-t path in the residual. Given such a path, we find the edge of minimum (residual) capacity along the path, and push this much flow along the path. Intuitively, whenever we find an augmenting path in $G_f(\Delta)$, we are guaranteed to push at least Δ units of flow, which means that we are guaranteed to increase our flow by at least Δ. We continue computing such augmenting paths (and updating the residual) until $G_f(\Delta)$ has no augmenting path. We then reduce Δ in half, and repeat the process. When $\Delta = 1$, we are effectively running the standard Ford-Fulkerson algorithm, which implies that we'll obtain the correct final flow. The algorithm is presented in the following code block below. Fig. 3 shows one iteration of the algorithm.

[2] Recall that a *cut* is a partition (X, Y) of the vertices such that $s \in X$ and $t \in Y$, and the *capacity* of a cut is the sum of capacities of edges going from X to Y. In the previous lecture we proved that the value of any flow cannot exceed the capacity of any cut.

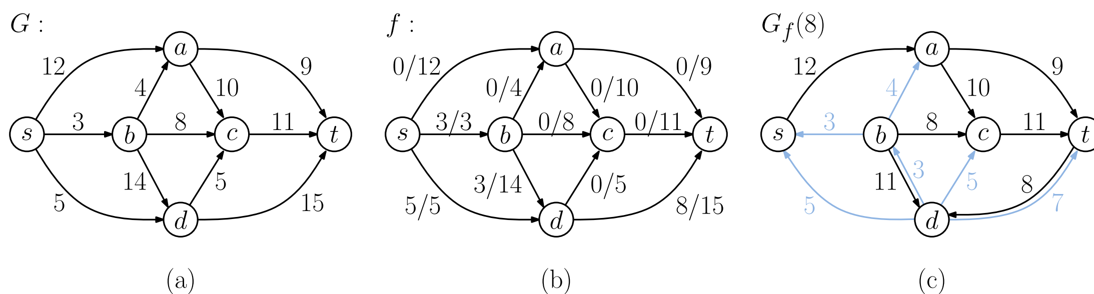

Fig. 2: (a) A network G, (b) a flow f, and (c) the restricted residual $G_f(8)$, which is G_f keeping only edges of weight ≥ 8. The light blue edges are the edges of G_f that have been removed.

Scaling Algorithm for Network Flow

```
scaling-flow(G) {                                  // network flow via scaling
    f = 0                                          // initial flow is zero
    C = sum of capacities out of s                 // capacity of any cut
    Del = 2^(floor(log_2(C))                       // largest power of 2 below C
    while (Del >= 1) {                             // repeat until resolution = 1
        R = residual network G_f                   // residual network
        remove from R edges of capacity < Del      // remove small capacity edges
        if (R has an s-t path P) {                 // augmenting path P?
            c = minimum capacity on P              // augmentation amount
            augment f by adding c to the flow on every edge of P
        } else {                                   // no more augmentations?
            Del = Del/2                            // reduce Delta by half
        }
    }
    return f                                       // return the final flow
}
```

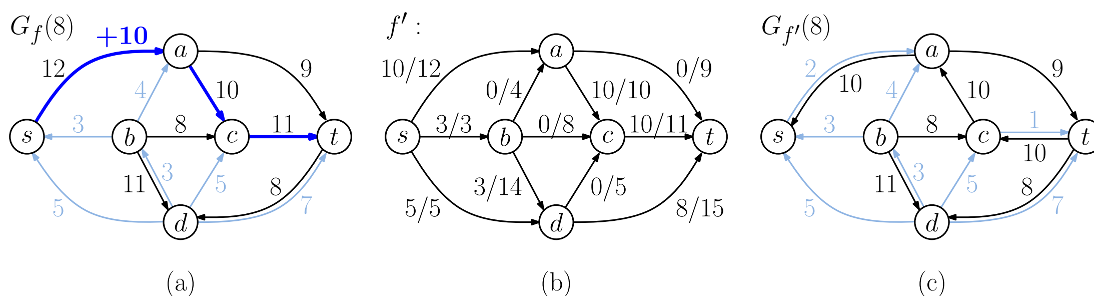

Fig. 3: (a) A single iteration of the scaling network-flow algorithm: (a) An s-t path in restricted residual network $G_f(8)$, (b) the flow f' that results by augmenting along this path, and (c) the updated restricted residual network $G_{f'}(8)$. This residual network has no s-t path, and so we would decrease Δ by half and continue.

Analysis of the Scaling Algorithm: We refer to the Kleinberg and Tardos book for a complete analysis of the scaling algorithm. Intuitively, the algorithm is efficient because each augmentation increases the flow by an amount of at least Δ. The minimum cut can have at most m edges. So, after $O(m)$ such augmentations, we will have effectively increased the flow along every edge of the minimum cut so much that its capacity in the residual network falls below Δ. When all the edges of the minimum cut disappear from $G_f(\Delta)$, it is not possible to augment further, and the algorithm goes on to the next smaller value of Δ.

Since each augmentation involves running BFS (or DFS) in $O(n+m) = O(m)$ time, it follows that after $O(m^2)$ time, we will exhaust augmentations in $G_f(\Delta)$, and will proceed to the next smaller value of Δ. After $O(\log C)$ halvings of Δ, we will have $\Delta < 1$, and the algorithm will terminate. Thus, the overall running time is $O(m^2 \log C)$. (It can be shown that a more efficient implementation runs in time $O(nm \log C)$.)

Pseudo-Polynomial and Strong Polynomial Time: Earlier, we complained that the Ford-Fulkerson algorithm is not really efficient, since its running time depends on the maximum flow value. The scaling algorithm also depends on the maximum flow value (albeit logarithmically rather than linearly). So, it what sense can we claim it is "efficient"?

Observe first that if the capacities are all small integers, then we can ignore the $\log C$ component, and so the running time is just $O(m^2)$ which is polynomial in the input size. (Efficient algorithms generally run in polynomial time, as opposed to exponential time.)

The algorithm is really *inefficient* when the capacities are *huge* numbers. Observe that if C is extremely large, then we require many bits to represent these numbers. Indeed, it takes $\Theta(\log C)$ bits to represent a number of magnitude C. Thus, if we think of the input size from the perspective of *total number of bits* needed to represent the input network, it can be shown that the scaling algorithm runs in time that is polynomial in this number of bits. (We will leave the details as an exercise.)

An algorithm whose running time is polynomial in the number of *bits* of input is called a *pseudo-polynomial time algorithm*. In contrast, an algorithm that runs in time that is polynomial in the number of *words* of input (such as n and m), is referred as running in *strongly polynomial time*. We will next show that there exist strongly polynomial time algorithms for network flow.

Edmonds-Karp/Dinic's Algorithm: As mentioned above, neither of the algorithms we have seen runs in strongly polynomial time, that is, polynomial in n and m, irrespective of the magnitudes of the capacity. Dinitz and (independently) Edmonds and Karp developed such an algorithm in the 1970's. (Note that Dinitz's algorithm goes by the name Dinic's algorithm, due to a mistranscription of his name.)

This algorithm uses Ford-Fulkerson as its basis, but with the minor change that the s-t path that is chosen in the residual network has the *smallest number of edges*. In particular, this just means that we run BFS in the residual network from s to t to compute the augmenting path. It can be shown that the total number of augmenting steps using this method is $O(nm)$.[3]

[3] This is not trivial to prove. A proof can be found in the algorithms book by Cormen, Leiserson, Rivest, and Stein. Intuitively, it can be shown that after at most m augmentations, each vertex's distance from s in the residual network increases by at least one edge, never to decrease again. Since a vertex's distance from s cannot exceed n, it follows that there are at most $O(nm)$ augmentations.

Since each augmentation takes $O(m)$ time to run BFS, the overall running time is $O(nm^2)$.

Even Faster Algorithms: The max-flow problem is widely studied, and there are many different algorithms. No one knows that what the lowest possible running time is for network flow, but a running time of $O(nm)$ has stood as an important milestone. See Table 1 for a summary of important results. The most recent (and most efficient) algorithms tend to be rather complicated.

Table 1: Running times of various network-flow algorithms, where $n = |V|$, $m = |E|$, C is any upper bound on the maximum flow.

Algorithm	**Year**	**Time**	**Notes**
Ford-Fulkerson (FF)	1956	$O(mC)$	Repeated augmentation
Gabow	1985	$O(nm \log C)$	FF + scaling
Edmonds-Karp	1972	$O(nm^2)$	FF + augment shortest paths
Dinic (Dinitz)	1970	$O(n^2 m)$	Blocking flows
Dinic + Tarjan	1983	$O(nm \log n)$	Dinic + better data structures
Goldberg and Tarjan	1986	$O(nm \log(n^2/m))$	Preflow push
King, Rao, Tarjan (KRT)	1994	$O(mn \frac{\log n}{\log(m/n \log n)})$	$O(nm)$ if $m = O(n^{1+\varepsilon})$
Orlin + KRT	2013	$O(nm)$	

Applications of Max-Flow: The network flow problem has a huge number of applications. Many of these applications do not appear at first to have anything to do with networks or flows. This is a testament to the power of this problem. In this lecture and the next, we will present a few applications from our book. (If you need more convincing of this, however, see the exercises in Chapter 7 of Kleinberg and Tardos. There are over 40 problems, most of which involve reductions to network flow.)

Maximum Matching in Bipartite Graphs: There are many applications where it is desirable to compute a pairing between two sets of objects. We present such a problem, called *bipartite matching* in the form of a "dating game," but the algorithm can be applied whenever it is desired to find pairing between objects of different classes subject to some compatibility criterion, for example, pairing medical students seeking residencies with hospitals.

Recall that an undirected graph $G = (V, E)$ is said to be *bipartite* if V can be partitioned into two sets X and Y, such that every edge has one endpoint in X and the other in Y. The pairing problem can be modeled as an undirected, bipartite graph whose vertex set is $V = X \cup Y$ and whose edge set consists of pairs (u, v), $u \in X$, $v \in Y$, such that u and v are can be paired (see Fig. 4(a)).

Given any undirected graph, a *matching* is defined to be a subset of edges $M \subseteq E$ such that for each $v \in V$, there is at most one edge of M incident to v. Our objective is to compute a matching in G that has the highest number of edges. Such a matching is called a *maximum matching*. Matchings can be computed in any undirected graph, but they are easiest to compute in bipartite graphs. This problem is called the *maximum bipartite matching problem* (see Fig. 4(b)).

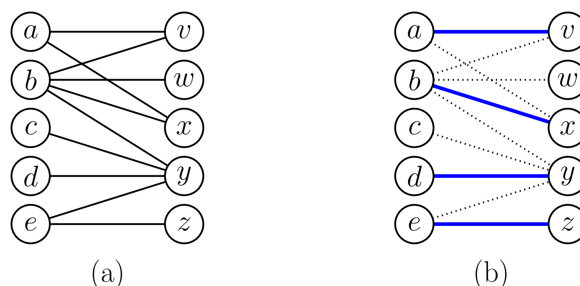

Fig. 4: (a) A bipartite graph G and (b) a maximum matching.

We will now present a reduction from maximum bipartite matching to network flow. In particular, we will show that, given any bipartite graph G for which we want to solve the maximum matching problem, we can convert it into an instance of network flow G', such that the maximum matching on G can be easily extracted from the maximum flow on G'.

To do this, we construct a flow network $G' = (V', E')$ as follows. Let s and t be two new vertices and let $V' = V \cup \{s, t\}$.

$$E' = \begin{cases} \{(s,u) \mid u \in X\} \cup & \text{(connect source to left-side vertices)} \\ \{(v,t) \mid v \in Y\} \cup & \text{(connect right-side vertices to sink)} \\ \{(u,v) \mid (u,v) \in E\} & \text{(direct } G\text{'s edges from left to right).} \end{cases}$$

Set the capacity of all edges in this network to 1 (see Fig. 5(b)).

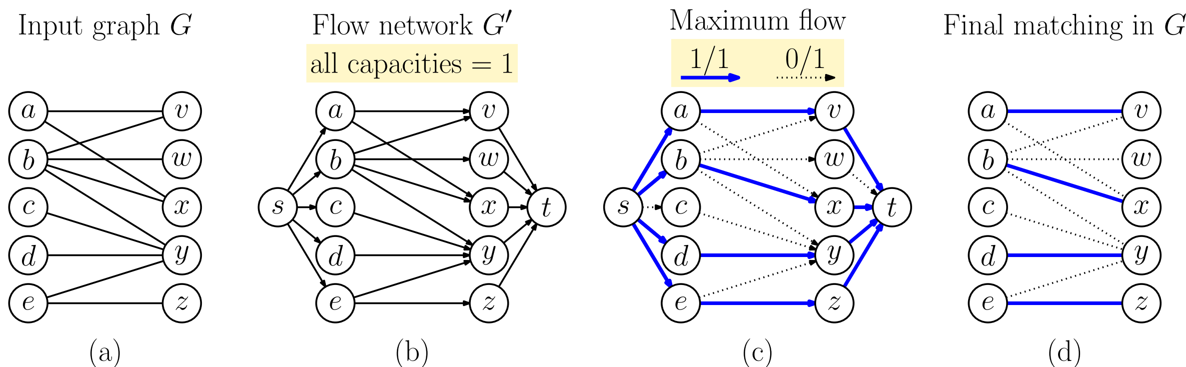

Fig. 5: Reducing bipartite matching to network flow.

Compute the maximum flow in G' (see Fig. 5(c)). The following lemma show that maximizing the flow in G' is equivalent to finding a maximum matching in G.

Lemma: Given a bipartite graph G, G has a matching of size k if and only if G' (constructed above) has a flow of value k.

Proof: ($\Rightarrow$) Let M denote any matching in G. We construct a flow in G' as follows. For each edge $(x, y) \in M$, set the flow along edges (s, x), (x, y), and (y, t) to 1. All the edges remaining edges of G are assigned a flow of 0. We assert that f is a valid flow for G'. By our construction, each vertex x receives one unit of flow coming in from s, sends one

unit to y, and y sends one unit to t. Therefore, we have flow conservation. Second, since M is a matching, each vertex of X or Y is incident to a single edge of M, which implies that it carries at most one unit of flow, which implies that the capacity constraints are all satisfied. Therefore, f is a valid flow in G'. By our construction, $|f| = |M|$.

($\Leftarrow$) Suppose that G' has a flow f. We may assume that this is an integer flow. Since all edges have capacity 1, it follows that the flow value on each edge is either 0 or 1.

Let M denote the edges of $X \times Y$ that are carrying unit flow in f. Observe that for every vertex of X, it has exactly one incoming edge (from s) of capacity 1, and hence it can be incident to at most one edge of M. Symmetrically, every vertex of Y has exactly one outgoing edge (to t) of capacity 1, and hence it also can be incident to at most one edge of M. Therefore, M is a matching in the original graph G. (An example is shown in Fig. 5(d)). Since each edge carries one unit of flow, the total value of the flow is the number of edges of M, that is, $|f| = |M|$.

Because the capacities are so low, we do not need to use a fancy implementation. Recall that Ford-Fulkerson runs in time $O(m \cdot C)$, where C is an upper bound on maximum flow. In our case C is at most n (the size of the largest possible matching). Therefore, the running time of Ford-Fulkerson on this instance is $O(nm)$.

By the way, there are other algorithms designed specifically for maximum bipartite matching. The best is due to Hopcroft and Karp. It runs in $O(\sqrt{n} \cdot m)$ time. There are also efficient algorithms for computing maximum matchings in general undirected graphs. The best known is the *blossom algorithm* of Jack Edmonds, which dates way back to 1961.

Summary: We have shown that the Ford-Fulkerson algorithm can take super-polynomial time to run. As alternatives, we introduced Gabow's scaling algorithm, which runs in time $O(nm \log C)$ and briefly discussed the Edmonds-Karp algorithm, which runs in $O(nm^2)$ time. Finally, we presented a reduction which shows that maximum matchings in bipartite graphs can be reduced to the network flow problem.