CMSC 451: Lecture 16
Network Flow Algorithms
Thursday, Nov 7, 2016

Reading: Sections 7.1, 7.3, and 7.5 in KT.

Algorithmic Aspects of Network Flow: In the previous lecture, we presented the Ford-Fulkerson algorithm. We showed that on termination this algorithm produces the maximum flow in an s-t network. In this lecture we discuss the algorithm's running time, and discuss 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 an integer amount. Thus, the resulting residual network also has integer capacities. Therefore, after a finite number of augmentations the algorithm must terminate.

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.

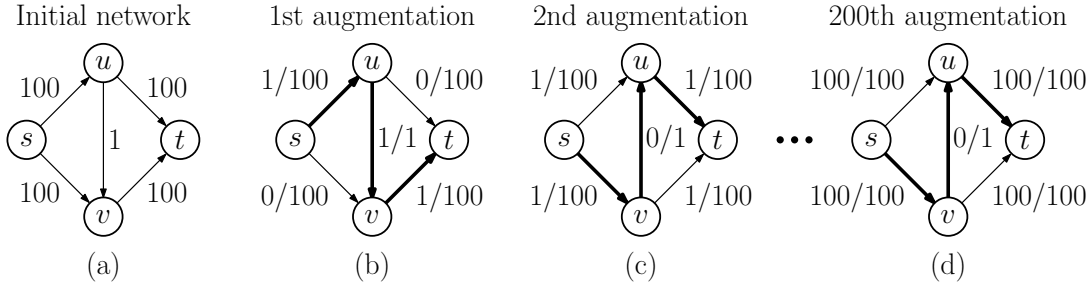

Fig. 1: Bad example for Ford-Fulkerson.

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

[1] This is not an unreasonable assumption. First, observe that if we multiply all the capacities by some positive constant c, all the flow values can be increased by this same factor. Assuming that the capacities are represented as fixed-point decimals with k digits to the right of decimal point, we can scale each capacity by $c = 10^k$, which converts all the capacities to integers. We solve this integer-capacity problem, and then divide the final result by c, thus mapping the solution back to the original instance.

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, the number of augmentation steps can be as high as $|f|$. If we make the reasonable assumption that each augmentation step takes at least $\Omega(m)$ time, the total running time can be as high as $\Omega(m|f|)$. Since $|f|$ may be arbitrarily high (it depends neither on n or m), this running time could be arbitrarily high, as a function of n and m.

Scaling Algorithm: In the above example, we made the apparently foolish decision to augment on a path of very *low* capacity. Can we do better by selecting paths of high capacity? We will see an example of such an algorithm, called the *scaling algorithm*. For the sake of efficiency, the algorithm does not augment along the path of highest possible capacity, merely a path of relatively high capacity.

The idea is to start with an upper bound on the maximum possible flow. The sum of capacities of the edges leaving s suffices, that is, $C = \sum_{(s,v)\in E} c(s,v)$. Clearly, the maximum flow value cannot exceed C. We initialize the scaling parameter Δ to be the largest power of 2, such that $\Delta \leq C$. Given any flow f (initially the flow of value 0), define $G_f(\Delta)$ to be the residual network consisting *only of edges of residual capacity at least* Δ. Since we only deal with integer capacities, when $\Delta = 1$, $G_f(\Delta)$ is the true residual network. 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 make good progress. Next, find an s-t path in $G_f(\Delta)$, augment the flow along this path, and update $G_f(\Delta)$ accordingly. We repeat the process until no augmenting paths remain. We then set $\Delta \leftarrow \Delta/2$ and repeat. When the value of Δ falls below 1, we terminate the algorithm and return the final flow. See the code block below and Fig. 2.

Scaling Algorithm for Network Flow

```
scaling-flow(G = (V, E, s, t)) {
    f = 0                                   // all edges carry zero flow
    D = largest power of 2 not larger than sum of capacities out of s
    while (D >= 1) {                        // when D = 1, Gf(D) has all edges
        Gf(D) = residual of G w.r.t. f, keeping only edges of capacity >= D
        while (there is an s-t path in Gf(D)) }
            P = any augmenting s-t path in Gf(D) // augment along the "fat" edges
            f' = augmenting flow for P
            f = f + f'
        }
        D = D/2                             // shrink D's value
    }
    return f                                // return the final flow
}
```

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

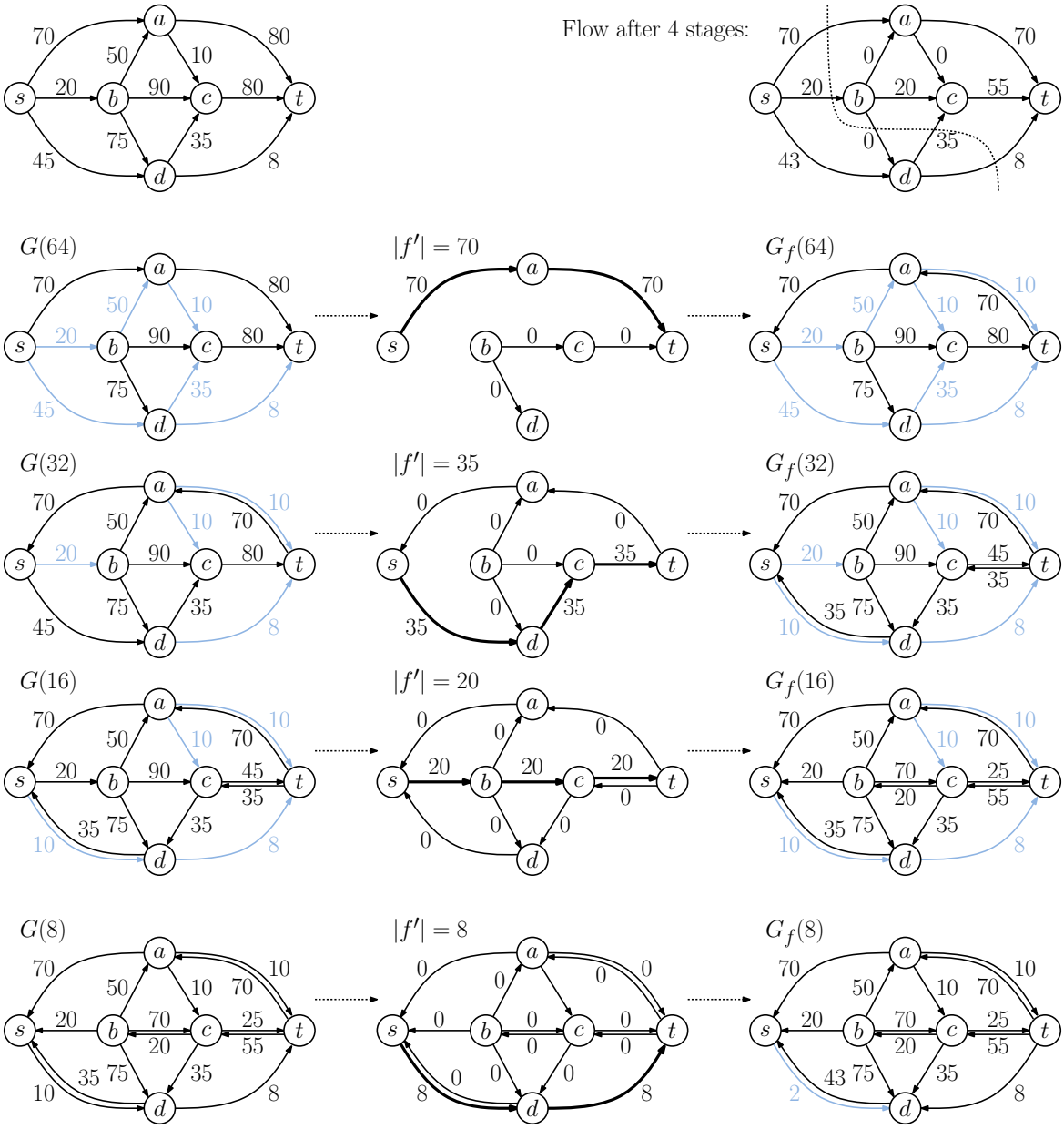

Fig. 2: The first four stages of the scaling algorithm for network flow. Note that at each stage, the value of the augmenting flow f' is at least Δ. (The algorithm will run for $G(4)$, $G(2)$, and $G(1)$, but no changes to the flow will occur.)

every edge of the minimum cut so much that its capacity in the residual graph 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 graph, 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*.

Edmonds-Karp 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. Edmonds and Karp developed such an algorithm in the 1970's (and it is claimed Dinic actually discovered this algorithm independently a couple years earlier).

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 graph from s to t to compute the augmenting path. It can be shown that the total number of augmenting steps using this method is $O(nm)$.[2] 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.

[2]This is not trivial to prove. Neither of our textbooks gives a proof, but one can be found in the algorithms book by Cormen, Leiserson, Rivest, and Stein. Intuitively, it can be shown that after at most m augmentations, some vertex's distance from s increases by one, never to decrease again. Since a vertex's distance from s cannot exceed n, it follows that there are at most $O(nm)$ augmentations.

After a long sequence of improvements, in 2013 it was shown that the problem can be solved in this time. The final algorithm is not very elegant. It is a hybrid of two different algorithms, one by King, Rao, and Tarjan (KRT) that runs in $O(nm)$ time for dense graphs and another by Orlin that runs in $O(nm)$ time for sparse graphs. Whether there exist a unified algorithm that achieves this bound or even faster algorithms remain as open research questions.

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

Algorithm	**Year**	**Time**	**Notes**
Ford-Fulkerson	1956	$O(mC)$	
Gabow	1985	$O(nm \log C)$	Scaling
Edmonds-Karp	1972	$O(nm^2)$	Ford-Fulkerson + augment shortest paths
Dinic	1970	$O(n^2m)$	Blocking flows in a layered graph
Dinic + Tarjan	1983	$O(nm \log n)$	Dinic + better data structures
Preflow push	1986	$O(nm \log(n^2/m))$	Goldberg and Tarjan
King, Rao, Tarjan	1994	$O(mn \log_{\frac{m}{n \log n}} n)$	$O(nm)$ if $m = O(n^{1+\varepsilon})$
Orlin + KRT	2013	$O(nm)$	Orlin: $O(nm)$ time for $m \leq O(n^{16/15-\varepsilon})$ KRT: $O(nm)$ for $m > n^{1+\varepsilon}$

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 KL. 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.

Suppose you are running a dating service, and there are a set of men X and a set of women Y. Using a questionnaire you establish which men are compatible which women. Your task is to pair up as many compatible pairs of men and women as possible, subject to the constraint that each man is paired with at most one woman, and vice versa. (It may be that some men are not paired with any woman.)

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. This 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 compatible (see Fig. 3(a)). Given a 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. Clearly, the objective to the

dating problem is to find a maximum matching in G that has the highest cardinality. Such a matching is called a *maximum matching* (see Fig. 3(b)).

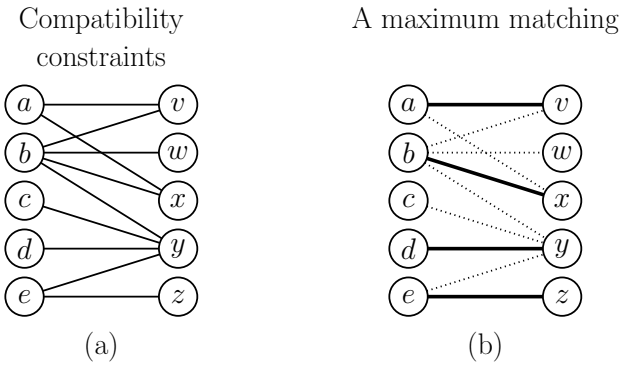

Fig. 3: A bipartite graph G and a maximum matching in G.

The resulting undirected graph has the property that its vertex set can be divided into two groups such that all its edges go from one group to the other. This problem is called the *maximum bipartite matching problem.*

We will now show a reduction from maximum bipartite matching to network flow. In particular, we will show that, given any bipartite graph G (see Fig. 4(a)) 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 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. 4(b)).

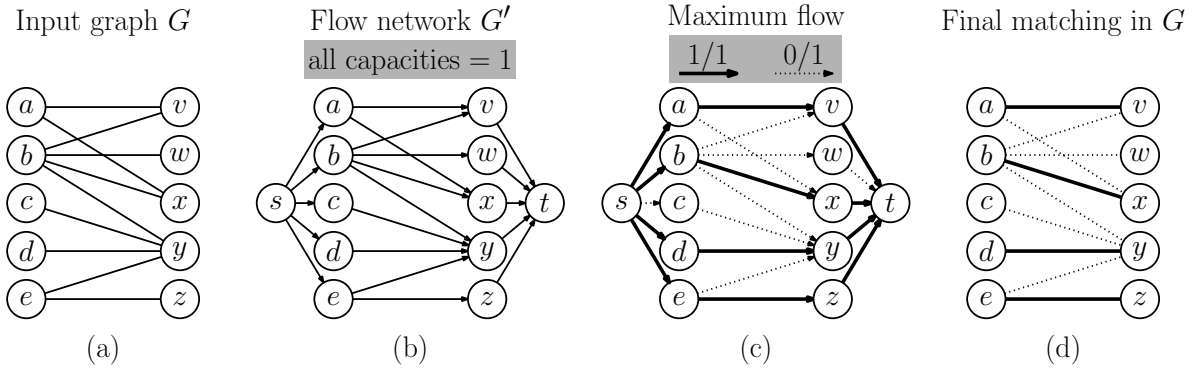

Fig. 4: Reducing bipartite matching to network flow.

Compute the maximum flow in G' (see Fig. 4(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 x if and only if G' (constructed above) has a flow of value x.

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. 4(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 $O(m \cdot F^*)$, where F^* is the final maximum flow. In our case F^* is at most n (the size of the largest possible matching). Therefore, the running time of Ford-Fulkerson on this instance is $O(m \cdot F^*) = O(nm)$.

There are other algorithms for maximum bipartite matching. The best is due to Hopcroft and Karp, and runs in $O(\sqrt{n} \cdot m)$ time.