CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 21

April 12, 2022

Instructor: Sepehr Assadi

1 Network Flow

We now study our final graph algorithms problem, the *network flow* problem (or the *maximum flow* problem). Generally speaking, the network flow problem captures the following scenario: we have a directed graph $G = (V, E)$, source vertex s, and a sink vertex t. We shall think of the source as producer of a certain material, or a "flow", at some fixed rate, and the sink as the consumer of this material. The edges of this directed graph are able to "transport" this flow subject to some given "capacity" on each edge. The question is then what is the "maximum rate" we can transport the materials from source to sink?

1.1 (Flow) Networks and Flows

Let us now define the network flow problem formally. A *network* (sometimes called a flow network) for our purpose is any *directed* graph $G = (V, E)$ with *capacity* c_e over each edge $e \in E$, and two designated vertices $s, t \in V$ where s is called a *source* vertex (and has no incoming edges) and t is called a *sink* vertex (and has no outgoing edges). Figure 1 gives an example of a network.

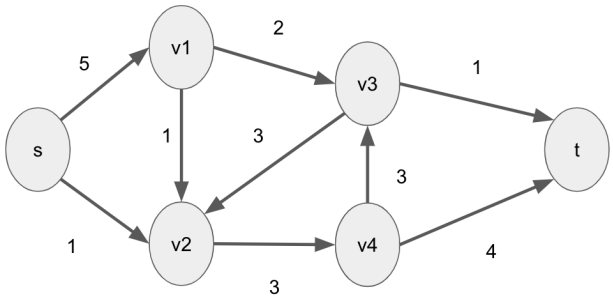


Figure 1: A flow network. Numbers next to each edge shows the capacity of the edge.

We define a *flow* in a given network $G = (V, E)$ as any function $f : V \times V \to \mathbb{R}^+$ that satisfies the following constraints:

- **Capacity constraints:** For any edge $e = (u, v) \in E$, $f(u, v) \leq c_e$ and if there is no edge from u to v, then $f(u, v) = 0$ (a flow over each edge cannot be larger than the capacity of the edge).
- **Preservation of flow:** For any vertex $v \in V - \{s, t\}$, $\sum_u f(u, v) = \sum_w f(v, w)$ (the flow into a vertex other than s, t is equal to the flow out of that vertex).

For any flow f in a network G, we define the *value* or *size* of the flow f to be $|f| = \sum_v f(s, v)$, i.e., the amount of flow produced by s (note that this is also equal to the amount of flow consumed by sink, i.e., $|f| = \sum_v f(v, t)$; why?). Figure 2 gives an example of a flow in the network of Figure 1.

We can now define the network flow (or maximum flow) problem as follows.

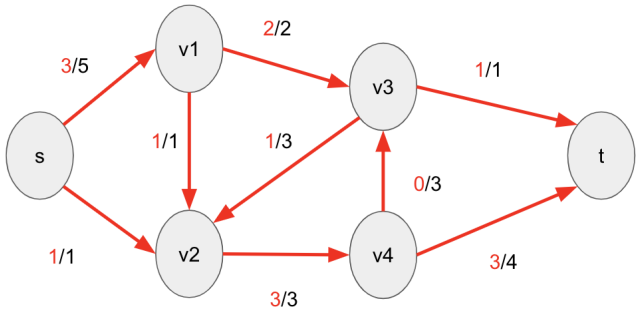


Figure 2: A flow function in the network given in Figure 1. Numbers next to each edge shows the value of flow function over each edge compared to the capacity of the edge. This flow is also a maximum flow.

Problem 1 (Network Flow/Maximum Flow). Given a network $G = (V, E)$ with source s and sink t and capacity c_e on every edge $e \in E$, find a flow function f with maximum value, namely a *maximum flow*.

1.2 Algorithms for Maximum Flow

There is a long list of algorithms designed over the years for solving the maximum flow problem. In fact, this is still a highly active area of research ~~and it seems that we are still far from finding the "best" (most efficient) algorithm for the maximum flow problem~~[1] – while the previous iterations of the course could say the above sentence, very recent progress on this problem (as of barely a month ago), has made substantial progress on this problem and at this point, up to lower order terms, the problem is more or less settled.

One of the earliest (perhaps the earliest) algorithms for this problem is the Ford-Fulkerson algorithm which finds a maximum flow in a network with n vertices and m edges in $O(m \cdot F)$ time where F denotes the value of maximum flow itself. Another algorithm for the maximum flow problem is the Edmond-Karp algorithm that runs in $O(m^2 \cdot n)$ time. Both of these algorithms are covered in your textbook and you are strongly encouraged to at least briefly take a look at these algorithms as they are not that complicated (albeit still considerably more involved than the previous algorithms we saw in this course). However, the more recent algorithms for this problem with improved running time are getting much more complicated and are entirely beyond the scope of this course.

Even though Ford-Fulkerson and Edmond-Karp algorithms are not that complicated (and are sometimes taught in undergraduate algorithm courses), in this course, we will *not* go over these algorithms. Instead, **we will use algorithms for maximum flow in a black-box manner using graph reductions to solve other problems**. As such, for the purpose of this course, we only need the following statement:

> **Ford-Fulkerson Algorithm:** There is an algorithm for the maximum flow problem that runs in $O(m \cdot F)$ time where F is the value of maximum flow in the network.

In addition, we need the following two properties of maximum flow found by the Ford-Fulkerson algorithm.

Integrality of Max Flows

Recall that we said a flow f is a function $V \times V \to \mathbb{R}^+$ which satisfies capacity constraints and preservation of flow. Note that importantly, this means the value of flow can be fractional on some edges. However, we so far always treated the maximum flow found from s to t in our graph to be *integral*, namely, the flow

[1] This is a very different scenario than shortest path and MST problems where we already know (for the most part) essentially the asymptotically best possible algorithms for those problems.

passed through every edge is an integer. But is this a correct assumption or can it be the case that in some networks the maximum flow should necessarily use some fractional values?

It turns out that as long as capacity of every edge is an integer, there always exists a maximum flow with integral values over every edge (clearly, if capacities of some edges are rational, we may need our maximum flow to also be rational). This statement requires a proof and follows from the proof of correctness of Ford-Fulkerson's algorithm. **However, for the purpose of this course, you may always assume that as long as capacities are integer, the maximum flow found by the algorithms is also integral.**

Cycle-Freeness of Max Flows

Another property of maximum flows we can always assume in this course is that the set of edges (u, v) with $f(u, v) > 0$ do *not* contain a (directed) cycle (even though the network may indeed have cycles in it) – this again requires a formal proof but we will skip that for the purpose of this course.

In this lecture and the next one, we will go over several applications of network flow such as *finding edge/vertex disjoint paths*, *bipartite matching*, or *exam scheduling* problems.

Application I: Finding Edge-Disjoint or Vertex-Disjoint Paths

Perhaps the simplest application of maximum flow is to find the *maximum number of edge-disjoint paths* from a vertex s to a vertex t in a given graph $G = (V, E)$. This problem is applicable in many settings when we want to have some "fault-tolerance" so that even if some edges are being removed (think of roads being blocked), we can still find a route from the source to the target.

Reduction. Simply turn G into a network by assigning capacity 1 to every edge (and if G is originally undirected, add both direction of edges to make it directed). Then find the maximum flow from s to t in this network and return the edges with non-zero flow as the edges of the paths.

Proof of correctness. Let us now prove its correctness.

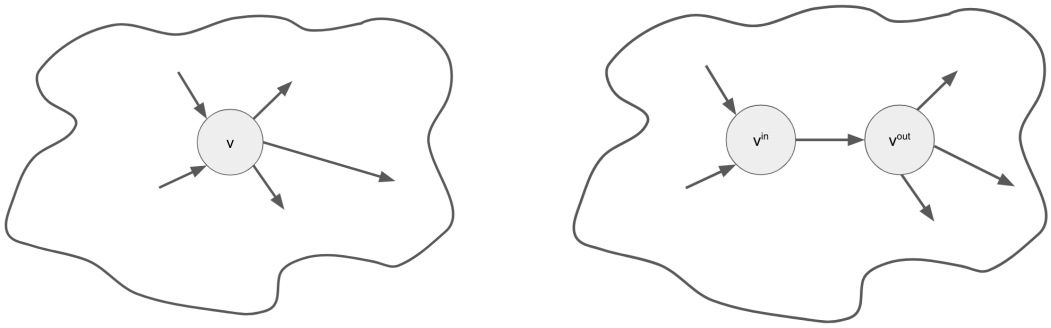
- Firstly, any s-t flow of value k in this network corresponds to a collection of edge-disjoint paths $P_1, \ldots, P_k$; this is because each edge can only carry one unit of flow in the network and so any of the k units of flow starting from s and ending in t should follow a path, edge-disjoint from the rest, from s to t. This proves that the solution is *feasible*.
- Secondly, any collection of edge-disjoint paths $P_1, \ldots, P_\ell$ in the graph implies a flow of value ℓ in the network; simply route one unit of flow across each path and since they are edge-disjoint, no edge of the network carry more than 1 unit of flow which is its capacity. This then means there is a one-to-one mapping between flows and edge-disjoint paths and thus by returning the maximum flow, we are also returning maximum size collection of edge-disjoint paths. This proves that the solution is *optimal*.

Runtime analysis. Creating the network takes $O(n+m)$ time and computing Ford-Fulkerson takes $O(m \cdot F)$ time, where F is the maximum value of flow from s to t. Since the value of such flow is at most $n-1$ (because at most $\deg(s) \leq n-1$ edges go out of s and any edge-disjoint path uses one of them), we have $F = O(n)$. So the runtime of this algorithm is $O(mn)$.

Important remark: We should *always* state the runtime of our algorithms in terms of parameters of the *original* input. So in this problem, even though we did a reduction to the maximum flow problem and ended up getting an algorithm with $O(m \cdot F)$ time, F is not a well-defined term in the context of the edge-disjoint path problem (it is a parameter of the solution not the input). So we should then switch F to a parameter governed by the input which in our case was $O(n)$. Hence, we state the runtime as $O(mn)$.

Vertex-disjoint paths? What if we are interested in finding vertex-disjoint paths from s to t (obviously, all these paths should still share s and t but beside s, t no other vertex should be used in more than one of

them)? A simple way is to introduce something like "vertex capacities" by doing the following: turn every vertex v in the original graph G into two vertices v^{in} and v^{out}, and turn any edge (u, v) in the original graph to an edge (u^{out}, v^{in}) (with capacity one); then also add the edge (v^{in}, v^{out}) with capacity one to the graph (see Figure 3).



(a) A single vertex in the original graph (b) The two copies of a vertex in the new network

Figure 3: An illustration of the reduction for finding vertex-disjoint paths.

By finding maximum flow in this network, we can find maximum number of vertex disjoint-paths from s to t (the proof is straightforward and is left as an exercise). This gives an $O(mn)$ time algorithm as before.

Note: This idea of introducing "vertex capacities" is quite general and can be used in other settings of network flows as well (we can add arbitrary capacity c_v to a vertex v by making the edge (v^{in}, v^{out}) have this capacity c_v).

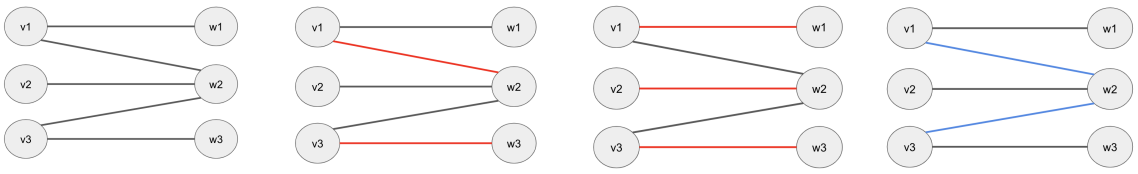
2 Application II: Bipartite Matching Problem

We now look at a different applications of network flows. Suppose we have a set of n buyers and a set of n items. Each buyer i is interested in buying exactly one item from a subset S_i of all n items and each item can be sold to at most one buyer. Can we find a way of selling all the n items to the n buyers? We can model this problem as the following graph problem:

Problem 2 (Bipartite Matching). We are given a graph $G = (V, E)$ where the vertices of V can be partitioned into two sets L and R such that every edge of G is between a vertex in L and a vertex in R (i.e., there are no edges with both endpoints in L or in R). Such a graph is called *bipartite*.

We define a *matching* M to be any subset of edges that *share no vertices*. A *maximum matching* is then any matching M with the largest number of edges. See Figure 4 for an illustration.

In the bipartite matching problem, we are given a bipartite graph $G = (V, E)$ and the goal is to find a maximum matching M of G.



(a) A bipartite graph (b) A matching (c) A maximum matching (d) Not a matching

Figure 4: An illustration of bipartite matching problem

The bipartite matching problem above captures the buyer-item scenario as follows: We create a bipartite graph $G = (V, E)$ with vertices in L corresponding to buyers and vertices in R corresponding to the items; we then connect every buyer-vertex i in L to every item-vertex in S_i in R. It is immediate to see that any matching in G corresponds to an assignment of (some of) items to (some of) buyers so that no item is given to more than one buyer and no buyer is given more than one item. Hence, by solving the bipartite matching problem over graph G, we can decide whether or not we can sell all the items to the buyers.

Bipartite matching is an extremely useful problem in practice, for instance in online advertising (which advertisement to show to which viewer), auctions (which items to sell to which bidder), etc.

In the next lecture, we see an algorithm for solving the bipartite matching problem using maximum flow.