The Maximum Flow Problem

The historically first and most fundamental network flow problem is the Maximum Flow Problem. Recall that we have already seen two other flow models: Minimum Cost Flow and Multicommodity Flow.

Model

Given a network with N nodes and links among them. We would like to transport some entity (for example, data) from a source node to a destination node so that it flows along the links. The goal is to determine the maximum possible amount of flow that can be pushed through the network, such that link capacity contraints are obeyed and at the intermediate nodes (also called transshipment nodes) the flow is conserved, i.e., whatever flows into an intermediate node, the same amount must flow out.

Let us review the input for the problem, the objective and constraints and then let us formulate it as a linear programming task.

Input data:

- The capacity of the $i \to j$ link is $C_{ij} \geq 0 \;\; (i, j = 1, \ldots, N)$.
- There are two distinguished nodes: a source node s, and a destination node d.

Remarks:

- The links are directed in this model, C_{ij} and C_{ji} may be different.
- If the link from i to j is missing from the network, then $C_{ij} = 0$. Thus, the capacities automatically describe the network topology, too.

Constraints:

- *Capacity constraint:* The flow on each link cannot exceed the capacity of the link.
- *Flow conservation:* The total outgoing flow of a node is equal to the total incoming flow of the node, except for the source and the destination.

Objective:

Find the maximum amount of flow that can be sent through the network from the source to the destination, such that the link capacity is not exceeded on any link and the flow conservation constraint is satisfied at every intermediate (transshipment) node.

LP Formulation

Let x_{ij} denote the flow on link (i, j). (The values of these variables are not known in advance, we want to optimize them.)

Let us express the constraints:

- The flow is nonnegative (by definition):

$$x_{ij} \geq 0 \qquad (\forall i, j)$$

- Capacity constraints:

$$x_{ij} \leq C_{ij} \qquad (\forall i, j)$$

- Flow conservation:

$$\sum_{j=1}^{N} x_{ij} - \sum_{k=1}^{N} x_{ki} = 0 \qquad (\forall i \neq s, d)$$

  Here the first sum is the total flow out of node i, the second sum is the total flow into node i.

The objective function is the total net flow out of the source:

$$F = \sum_{j=1}^{N} x_{sj} - \sum_{k=1}^{N} x_{ks}$$

Thus, the LP formulation is:

$$\max F = \sum_{j=1}^{N} x_{sj} - \sum_{k=1}^{N} x_{ks}$$

subject to

$$\begin{aligned} \sum_{j=1}^{N} x_{ij} - \sum_{k=1}^{N} x_{ki} &= 0 \qquad (\forall i \neq s, d) \\ x_{ij} &\leq C_{ij} \qquad (\forall i, j) \\ x_{ij} &\geq 0 \qquad (\forall i, j) \end{aligned}$$

Solution

Solving the Maximum Flow Problem directly as a linear program, although possible, is not economical. There are several special algorithms which utilize the special structure of the problem and, therefore, are faster and simpler than a general LP solution.

The historically first and most fundamental maximum flow algorithm is due to Ford and Fulkerson. Here we provide an informal summary on how the Ford-Fulkerson algorithm works.

The key idea of the Ford-Fulkerson algorithm is that if we can find a so called *augmenting path*, then the current flow (which may be initially 0) can be increased without violating any constraints.

An *augmenting path* is any undirected path from s to d, such that on any forward edge (an edge whose direction agrees with the path traversal) the

flow is strictly less than the capacity, while on any backward edge (an edge with opposite direction to the path traversal) the flow is strictly positive.

If such an augmenting path is found then we can increase the net flow out of the source, without constraint violation, by the following operation:

- Increase the flow on every forward edge by some $\epsilon > 0$ (the same ϵ for each edge).
- Decrease the flow by ϵ on every backward edge (using the same ϵ as above).
- Leave the flow unchanged on every edge not on the path.

One can directly check that with this operation we do not violate any constraint, assuming ϵ was chosen small enough, such that we do not go over the capacity on any forward edge, and do not pull the flow into negative on any backward edge.

While the augmenting operation keeps all constraints satisfied, the net flow out of the source increases by ϵ. The reasons are:

- If the first edge on the augmenting path is a forward edge, then the flow on this edge grows by ϵ, and it does not change on any other edge adjacent with the source. As a result, ϵ more flow is flowing out of the source.
- If the first edge on the augmenting path is a backward edge, then the flow on this edge decreases by ϵ, and does not change on any other edge adjacent with the source. As a result, ϵ less flow is flowing back into the source. Therefore, the *net* flow out of the source still grows by ϵ.

Thus, the augmenting operation increases the *net flow* that leaves the source, while keeping all constraints satisfied. This increased flow must eventually reach the destination, since it is preserved at every intermediate node.

Remark. It may seem at first an appealing interpretation of the augmenting operation that we "push through" more flow through the augmenting path. This is, however, not correct. It may even happen that the flow is reduced on every edge of the path, if they are all backward edges. Then how can the flow always increase? The answer is that the augmenting operation *rearranges* the flow in a way that the net flow out of the source eventually grows, but this does not necessarily mean that the increment is simply realized along the augmenting path.

Having introduced the augmenting path concept, there are some natural questions that arise at this point:

1. What is a good choice for ϵ?
2. How do we find an augmenting path?
3. What if there is no augmenting path?
4. How many iterations (augmentations) are needed?

Let us present the answers for each:

1. The value of ϵ should be small enough, so that we do not overshoot the capacity on any forward edge, and do not pull the flow into negative on any backward edge. Let us define the *residual capacity* of an edge (i, j) on the augmenting path as follows:

$$r_{ij} = \begin{cases} C_{ij} - x_{ij} & \text{if } (i,j) \text{ is a forward edge} \\ x_{ij} & \text{if } (i,j) \text{ is a backward edge} \end{cases}$$

   Thus, r_{ij} is the maximum flow increment allowed on the edge, if it is a forward edge, or the maximum flow decrement, if it is a backward edge. The reason to call it *residual capacity* is that this represents the

maximum remaining allowed change in the flow (increase or decrease, depending on whether the edge is forward or backward).

Then the right value of ϵ is the minimum of the r_{ij} values along the augmenting path. Why? Because we want ϵ as large as possible (so that we gain the largest possible flow increment by the augmentation), but ϵ must fit into each gap along the path, so that we do not violate the capacity and non-negativity constraints.

2. An augmenting path, if exists, can be found by a shortest path computation in an auxiliary graph G', called *residual graph*. This is easily constructed from the original, using the current flow, as follows. Let the residual graph have the same number of nodes as the original. Let us index them such that for each original node i we assign a node denoted by i' in the residual graph. Look now at each edge (i, j) in the original graph, and assign edges between i' and j', as follows:

   - If $x_{ij} = 0$, then include the edge (i', j') in G'.
     Reason: this can be a forward edge on an augmenting path (because the flow can be increased on it), but not a backward edge (because the flow cannot be decreased on it).
   - If $x_{ij} = C_{ij}$, then include the edge (j', i') in G'.
     Reason: this can be a backward edge on an augmenting path (because the flow can be decreased on it), but not a forward edge (because the flow cannot be increased on it).
   - If $0 < x_{ij} < C_{ij}$, then include the both edges (i', j') and (j', i') in G'. Reason: it can be either a forward or a backward edge on an augmenting path.

   With this construction it is easy to see that any directed $s' \to d'$ path in G' corresponds to an $s \to d$ augmenting path in the original graph. Thus, we only need to find a directed $s' \to d'$ path in G' (e.g., by

Dijsktra's algorithm), and then map back the path into the original graph.

3. Having found an augmenting path, we can increase the flow, without violating any constraint. But what if no augmenting path exists? Answer: in that case, the flow is necessarily maximum. This can be proved by the following simple, but ingenious reasoning:

   - Let A be the subset of nodes that are reachable from s via an augmenting path, with respect to the current flow. The source itself is also in A. Let B contain the remaining nodes. Observe that $d \in B$, because we look at the situation when there is no $s \to d$ augmenting path, so, by definition d cannot be in A.
   - Look at the edges connecting A and B.
     - On any edge $e = (i, j)$ with $i \in A$ and $j \in B$, the flow must be *equal* to the capacity. The reason is that, by definition, i is reachable from s via an augmenting path. If e is not saturated, then it could be added to this path as a forward edge, creating an augmenting path from s to j. This is, however, impossible, as $j \in B$, and B contains those nodes that are not reachable from s via an augmenting path.
     - On any edge $e = (k, \ell)$ with $k \in B$ and $\ell \in A$, the flow must be *zero*. The reason is similar as above: the node ℓ, being in A, is reachable from s via an augmenting path. If e has positive flow, then it could be added to this path as a *backward* edge, creating an augmenting path from s to k. This is, however, impossible, as $k \in B$, and B contains those nodes that are not reachable from s via an augmenting path.

   Thus, we can conclude that the edges that go from A to B must be all saturated, and the ones that go from B to A do not carry any flow. This means, the current flow saturates the *cut* defined

by A and B: all edges from A to B carry the maximum possible amount of flow (their capacity), and nothing is flowing back.

Observe now that every such cut is a bottleneck: we cannot push through more flow than the capacity of the cut, which is defined as the summed capacity of all edges that go from the source side to the destination side.

Thus, having reached a bottleneck, the flow must be maximum, since no more flow can be pushed through than the capacity of any cut. Since we can only reach the smallest bottleneck, as it already limits how high we can go, therefore, the flow has reached its maximum, once we find no more augmenting paths. As a by-product, we obtain an important result, the famous Max Flow Min Cut Theorem:

Theorem 1 *The value of the maximum flow from s to d is equal to the minimum capacity of any cut which separates d from s.*

Remark. Since no more flow can be pushed through any cut than its capacity, therefore, the total $s \to d$ flow can never be larger than the capacity of *any* $s \to d$ cut (a cut that separates d from s). That is, for any feasible value F of the $s \to d$ flow and for any $s \to d$ cut with capacity C

$$F \leq C$$

always holds. The nice thing is that if we maximize the lefthand-side and minimize the righthand-side, then we get precise equality. In other words, whatever is *not obviously excluded* by the capacity limitations, *can actually be achieved.* There is no gap here between the theoretically possible and the practically achievable!

4. Regarding the number of iterations (augmentations) the algorithm has to do in the worst case, it can happen that with unlucky choices for the augmenting paths, the number of iterations can grow exponentially.

On the other hand, one can easily avoid this, as shown in the following theorem.

Theorem 2 *If among the possible augmenting paths in the residual graph a shortest path, using the residual capacities as edge weights, is chosen in every iteration, then the algorithm reaches the maximum flow after at most $nm/4$ iterations, where n is the number of nodes and m is the number of edges in the graph.*

Note: This variant of the Ford-Fulkerson method is often called Edmonds-Karp algorithm.

Yet another important result is about the *integrality* of the maximum flow.

Theorem 3 *If all capacities are integers, then there exists a maximum flow in which the value of the flow on each edge is an integer (and, therefore, the total flow is also integer). Such an integer flow can also be efficiently found by the Ford-Fulkerson (or Edmonds-Karp) algorithms.*

This theorem is not hard to prove: if we start with the all-zero flow, and all capacities are integers, then one can prove with a simple induction that ϵ can always be chosen an integer, so the flow remains integer after each augmentation.

The integrality theorem makes it possible to use maximum flow computations for solving graph optimization problems. Key examples are the maximum number of disjoint paths, and the maximum bipartite matching problem. See details in the slide set entitled "Application of Maximum Flow to Other Optimization Problems," as well as in the class discussion.