02-613 Week 12

Algorithms and Advanced Data Structures

Aidan Jan

November 14, 2025

Proving Min Cut Max Flow

We covered max flow and the Ford-Fulkerson Algorithm previously. Now, we can define a cut through a graph. A cut in undirected $G = (V, E)$ divides G into partitions A and $B = V \setminus A$. The **weight** of a cut is

$$\sum_{a \in A, b \in B, (a,b) \in E} w(a, b)$$

An st-cut is a cut with $s \in A, t \in B$. The **capacity** of a cut is

$$\sum_{a \in A, b \in B, (a,b) \in E} c(a, b)$$

The concept of having a *min* cut is important because it is the same problem as max flow. We can show this with a quick proof.

Theorem 1: let f be an st flow, and let (A, B) be an st cut. Then, the value of the flow

$$v(f) = f^{out}(A) - f^{in}(A)$$

Proof 1: any flow that leaves A must either return or flow to t.

Theorem 2: let f be an st flow, and let (A, B) be an st cut. Then,

$$v(f) \leq \text{capacity}(A, B)$$

Proof 2:

$$\begin{aligned} v(f) &= f^{out}(A) - f^{in}(A) \\ &\leq f^{out}(A) \\ &= \sum_{e \in \text{cut}(A)} f(e) \\ &\leq \sum_{e \in \text{cut}(A)} c(e) - \text{capacity}(A, B) \end{aligned}$$

Now, consider f^* returned by Ford-Fulkerson, and let G_{f^*} be the residual graph for f^*. Then,

$$V(f^*) = \text{capacity}(A^*, B^*) = V(\hat{f})$$

From this, we get that the value of the max flow in any flow graph is equal to the capacity of the minimum cut.

The **Edmunds-Karp** algorithm is the one which solves min cut. This one can be run in $\mathrm{O}(nm^2)$.

Reductions to Max Flow

A reduction rewrites a problem in terms of another. To "reduce" A into B means getting a solution for B leads to a solution to A. We will be using this technique to solve many problems.

Maximum Bipartite Matching

Maximum bipartite matching is, given a bipartite graph $G = L \cup R, E$, find $S \subseteq E \subseteq L \times R$ that is as large as possible. What this essentially means, is imagine each node on the left being a person, and each node on the right being a task. The edges connecting them represent the person knows how to do the task. We want to assign tasks such that as many tasks as possible may be completed.

To reduce this to max flow, we can add a source node to all the nodes in L, and a sink to all the nodes in R. Let all the edges in the entire graph (including from source, and to sink), have a capacity of 1. This is because every node on the left needs to be paired to a node on the right. Now we can call Ford-Fulkerson on this.

Now, we have to explain how it is correct. Basically, Ford-Fulkersons stops running when there are no more valid paths from the source to the sink, which implies that as many tasks as possible must be taken up. If there was a pairing that could have been added, then Ford-Fulkerson's would have found it, so having a pairing not found leads to a contradiction.

Finally, we can extract the matching from Ford Fulkerson by looking at which connections between the left and right are filled.

The runtime of this algorithm is the runtime of Ford-Fulkerson's, which would be $O(m^2)$. In our graph, we have $n+2$ vertices (matching, plus source and sink), $m+n$ edges (since we added n edges connecting the source to all 'left' vertices and sink to all 'right' vertices), and a maximum cost of $m+n$. Since the runtime of Ford Fulkerson is $O(|E||C|)$, our runtime would be $O((m+n)(m+n)) = O(m^2)$.

Supply and Demand

Let G be a graph where we have supply nodes, and demand nodes. Edges have a capacity which represents the maximum product that could travel through the edge. We want to decide whether or not all the demand can be filled.

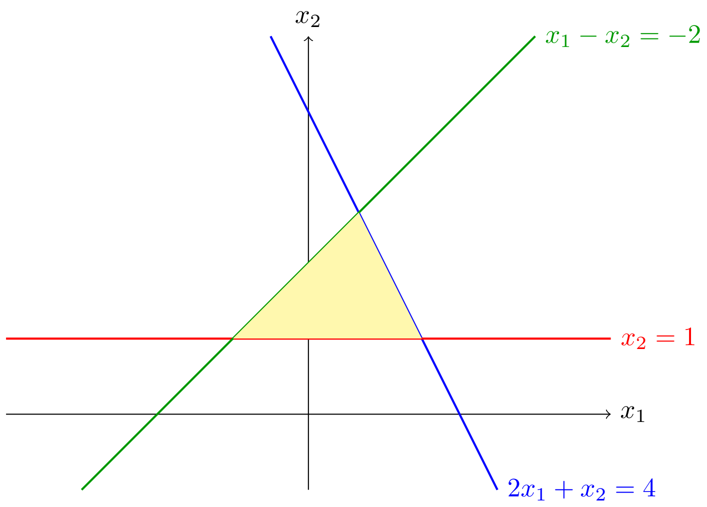
- First, compare the demand and the supply. If the total supply is less than the total demand, we can't.
- Next, we can add a source node connecting to all the supply nodes, where the edges have capacity of the supply node's supply. Similarly, we can add a sink node connecting to all the demand nodes which the edges added are the demand.
- Now, we can run Ford Fulkerson.
- The maximum demand we can satisfy would be the result of Ford Fulkerson.

Linear Programming

A **linear program** is a program that takes in constaints (linear equations), and finds values of variables that minimize an objective function. For example, one such program may be able to take in the constraints of two variables, $\{x_1, x_2\}$:

- $2x_1 + x_2 \leq 4$
- $x_2 \geq 1$
- $x_1 - x_2 >= -2$

Our objective function may be to minimize $-x_1 - x_2$.



From here, the constraints makes our search space within the triangle, so we just find the point in the triangle that minimizes $-x_1 - x_2$.

Examples of Linear Solvers

- CPLEX
- Gurdoi
- Simplex
- Ellipsoid
- Interior Points

Why is this important?

It turns that most of the problems covered before in this class can be written as linear programs. For example, given a flow graph $G = (V, E)$ with capacities c_e, find a flow f_e from $s \in V$ to $t \in V$ of max value, and each edge costs p_e per unit. In this case, we can have our objective to be to maximize

$$\sum_{e \in E} f_e \cdot p_e$$

with constraints

- $0 \leq f_e \leq c_e$
- $\sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w), \forall v \in V, v \neq s, t$

We can then turn this into input for a linear program, and we get the solution to the problem without using Ford Fulkerson's.

Example: Shortest Path

Given a directed graph $G = (V, E)$ with positive edge weights w_k, find the shortest path from $s \to t$. Let

$$x_{uv} = \begin{cases} 1 & \text{if } (u, v) \text{ is in shortest path} \\ 0 & \text{otherwise} \end{cases}$$

The goal is to minimize $\sum x_{uv} \cdot w_{uv}$, while subject to

- $\sum_{w:(u,t)} x_{ut} = 1$
- $\sum_{w:(s,v)} x_{sv} = 1$
- $\sum x_{uv} - \sum x_{vw} = 0 \quad \forall v \in V$
- $x_{uv} \in \{0, 1\}$

This can now be input into an LP. There *is* an important caveat though, which is here we are using integers. It turns out that an LP can run in polynomial time, but an Integer LP (ILP) is NP-Complete.