Algorithms and Probability (FS2025)
Week 12

Rui Zhang

May 15, 2025

Contents

1 Exercise Feedback **1**

2 Content **1**

2.1 Flow . . . 1

2.2 Flow Algorithm . . . 4

2.3 Applications of Flow . . . 6

2.3.1 Finding bipartite Matchings . . . 6

2.3.2 Menger's Theorem . . . 6

2.3.3 Image Segmentation . . . 7

1 Exercise Feedback

- The hint from exercise b) was super misleading and often, the case $\sum_{i\in[n]}(\mathbb{E}[\varepsilon_i^4]F_i^4)$ was counted too often.
- Try to remember lemmas / theorems to reduce computations, such as linearity of expectation or $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$ for independent X and Y

2 Content

2.1 Flow

This topic is the another very big part of this course. Flow algorithms are one of the hardest problems just still solvable in polynomial time. They are often used for Image processing and modeling traffic. Personally, when I think of flow, I always think of an extension to graphs, which model water flowing through pipes from one shared starting point and one shared ending point. And that is exactly how a 'network' is defined:

Definition: A Network is a tuple $N = (V, A, c, s, t)$, where

- (V, A) is a directed graph (where the edges are the pipes and the vertices are the intersections)
- $s \in V$ is the source (the point where water magically spawns out of)
- $t \in V \setminus \{s\}$ is the target (the point where water disappears into the abyss)
- $c : A \to \mathbb{R}_0^+$ is the capacity function (which denotes how much water can flow through each pipe)

You can think of a network as a glorified weighted, directed graph. However, don't tell anyone I said that. Now, we want to model the idea of water flowing through the pipes. We want to impose the conditions that water cannot disappear and appear at any point in the network except for at the source and at the target. Thus we get:

Definition: Let $N = (V, A, c, s, t)$ be a network. A flow in N is defined as a function $f : A \to \mathbb{R}$, such that

- For all $e \in A$, $0 \leq f(e) \leq c(e)$
- For all $v \in V \setminus \{s, t\}$ we have

$$\sum_{u \in V \text{ s.t. } (u,v) \in A} f(u, v) = \sum_{u \in V \text{ s.t. } (v,u) \in A} f(v, u)$$

  which we call conservation of flow (note that again, this does not apply for the source and the target)

Definition: The value of a flow f is defined as

$$\text{val}(f) := netoutflow(s) := \sum_{v \in V \text{ s.t. } (s,v) \in A} f(s, u) - \sum_{u \in V \text{ s.t. } (u,s) \in A} f(u, s)$$

Definition: f is called an integer flow, iff $f(e) \in \mathbb{Z}\ \forall e \in A$

Now, intuitively, since no amount of water appears / disappears at any point in the network except for at the source and the target, we must have that the amount of water appearing at s is equal to the amount of water disappearing at t:

Lemma:

$$\text{netinflow(t)} := \sum_{u \in V \text{ s.t. } (u,t) \in A} f(u, t) - \sum_{v \in V \text{ s.t. } (t,v) \in A} f(u, s) = \text{netoutflow}(s)$$

Proof. We can alternatively prove that

$$\begin{aligned}
0 &= \text{netoutflow}(s) - \text{netinflow}(t) \\
&= \left(\sum_{v \in V \text{ s.t. } (s,v) \in A} f(s,u) - \sum_{u \in V \text{ s.t. } (u,s) \in A} f(u,s) \right) \\
&\quad - \left(\sum_{u \in V \text{ s.t. } (u,t) \in A} f(u,t) - \sum_{v \in V \text{ s.t. } (t,v) \in A} f(u,s) \right) \\
&= \sum_{v \in V} \left(\sum_{u \in V \text{ s.t. } (v,u) \in A} f(v,u) - \sum_{u \in V \text{ s.t. } (u,v) \in A} f(u,v) \right) && (\star) \\
&= \sum_{(v,u) \in A} f(v,u) - \sum_{(u,v) \in A} f(u,v) && (\star\star) \\
&= 0
\end{aligned}$$

Here, we have to understand $(\star)$ and $(\star\star)$ better: $\star$ follows from the fact that the net- out and inflow values of all vertices v except for s and t are 0. $\star\star$ is just a rewrite: Instead of summing over all ingoing / outgoing edges, we sum over all vertices and count all ingoing / outgoing edges. □

From here on out, out goal is to derive the maxflow-mincut theorem - the intuition that the maximum flow (amount of water) that you can pump throught the pipes in a network N, is the same as the minumum bottleneck within the network. Right now, it might sound like a theorem that does not help that much. However, you will see from a few funny code experts the next few weeks that it can get very interesting. To make this intuition more formal, let us define what a cut even is:

Definition: An s-t-Cut for a network (V, A, c, s, t) is a partition (S, T) of V $(S \uplus T = V)$ such that $s \in S$ and $t \in T$. The capacity of this cut is defined as

$$\text{cap}(S,T) := \sum_{(u,w) \in (S \times T) \cap A} c(u,w)$$

We have that

Lemma: Let f be a flow and (S, T) an s-t-Cut in a network. Then we have that

$$\text{val}(f) \leq \text{cap}(S,T)$$

This gives us a "certificate" for maximum flows: If our flow has the same value as the capacity of some partition, then this flow is maximal.
With these definitions also comes the formal version of the

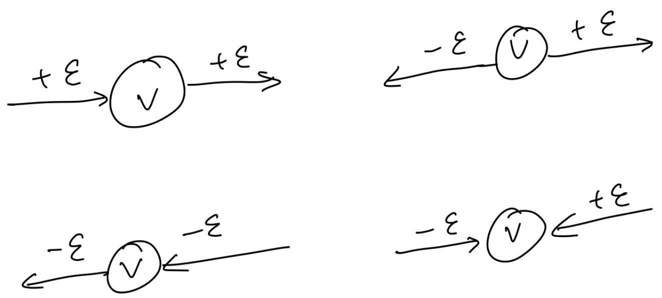
Definition: (Maxflow-Mincut-Theorem) For every network (V, A, c, s, t), we have that

$$\max_{f \text{a flow in } N} \text{val}(f) = \min_{(S,T) \text{ } s\text{-}t\text{-Cut in } N} \text{cap}(S,T)$$

Which we will prove partially via an algorithm:

2.2 Flow Algorithm

Given an existing flow, we have four ways of changing the flow at a vertex v in hopes of increasing the total flow:



Given a chain of such situations from source s to target t, we have a way of increasing the flow! This path we will also call an augmenting path (*war flashbacks to matchings) To formalize this, we will introduce the residual network.

Definition: Let $N = (V, A, c, s, t)$ be a network without edges in opposite directions and f be a flow in N. The residual network N_f is defined as (V, A_f, r_f, s, t), where

- If $e \in A$ with $f(e) < c(e)$ in the original network, then e is also an edge in A_f in the residual network with $r_f(e) := c(e) - f(e)$
- If $e \in A$ with $f(e) > 0$, then there is the edge e^{opp} in A_f with $r_f(e^{opp}) := f(e)$

There are no other edges in the network. We call $r_f(e)$ the residual capacity of e.

This residual network models exactly the situations mentioned above. Every s-t-path in the residual network corresponds to an augmenting path in the original network. We will augment by a path by finding the smallest residual capacity ε and augmenting the flow and increasing it along that path.

What you have proven in class is that

Theorem: A flow f in a network N is a maximum flow if and only if there are no s-t-paths in the residual network N_f anymore.

Proof. (sketch) One side of the implication ($\Leftarrow$) follows via an indirect proof: If there is an s-t-augmenting-path, then we can augment and get a bigger flow. What we have to prove for the other direction is essentially show that there exists a partition (S, T) which has the same capacity as val(f) (thus showing f is maximum) We do this by construction:

- Take S as the set of vertices reachable from s in the residual network. By assumption, $t \notin S$.
- If we take $T = V \setminus S$, then (S,T) is a valid s-t-cut.

For this cut, we have that the total flow from S to T is exacltly the capacity $\text{cap}(S,T)$ and that the total flow from T to S is 0. This gives us (based on the lemma that $val(f) = f(S,T) - f(T,S)$)

$$val(f) = f(S,T) - f(T,S) = \text{cap}(S,T) - 0 = \text{cap}(S,T)$$

□

Example 1.

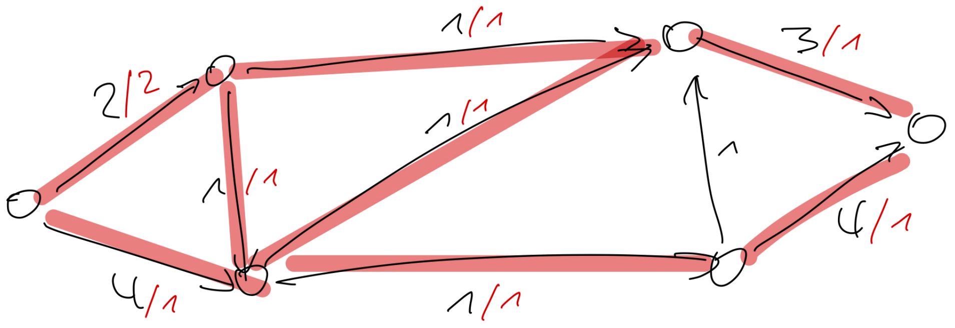


Figure 1: Take this network as an example and construct the residual network. Notice that there are no more s-t-paths in the residual network, because it is maximum

```
Algorithm 1 Ford-Fulkerson($V, A, c, s, t$)
1: $f \leftarrow \mathbf{0}$
2: while $\exists$ s-t-path P in $(V, A_f)$ do
3:     Increase the flow along P (find smallest residual capacity along P, augment)
4: end while
5: return $f$
```

This algorithm always terminates for integer networks, since each iteration, we will increase by at least an integer $\varepsilon > 0$ and the maximum flow is upper bounded. Now with this knowledge, we know that for all integer networks, we have an algorithm (Ford-Fulkerson) which finds us a flow f that has the value equal to a minumum cut, proving the max-flow-min-cut theorem (for integer networks).

Theorem: Let U be the maximum capacity over all edges in a network N. The runtime of Ford-Fulkerson in an integer network is $O(mnU)$ in this network N

Proof. Each iteration of Ford-Fulkerson will take $O(m)$ due to the construction of the residual network. We will have at most $O(nU)$ iterations, as each iteration we increase the flow by at least an integer $\varepsilon > 0$ and our flow is upper bounded by $O(nU)$ (the maximum number of outgoing edges s can have is n) □

Example 2.

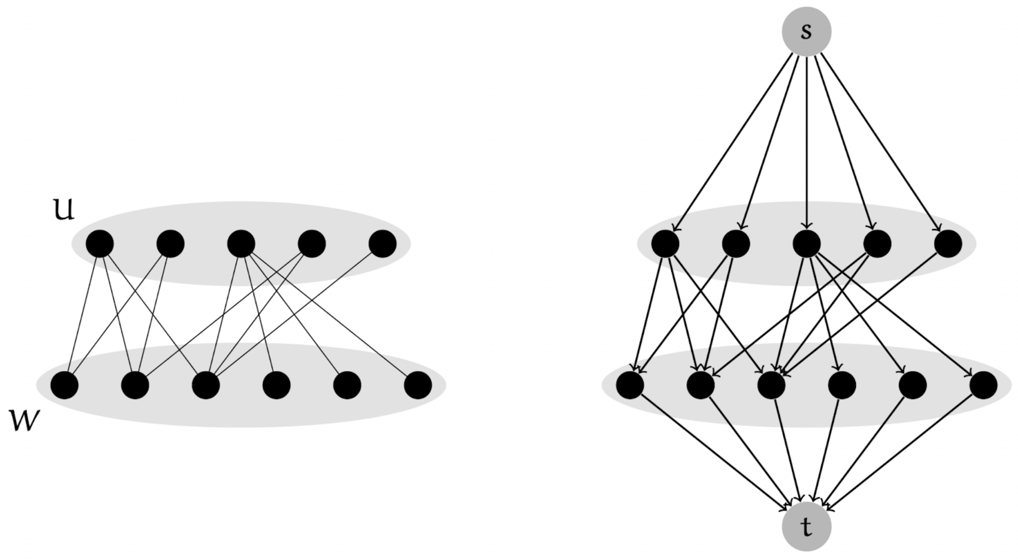


Figure 2: Construction of a network to find bipartite matchings.

2.3 Applications of Flow

Flow algorithms are often used in computer networking. Since none of you have taken the CN course yet, we will instead look at other problems we could solve using flow:

2.3.1 Finding bipartite Matchings

Given a bipartite graph, we would like to find the maximum matching (Kardinalitätsmaximal). Although less efficient than Hopcroft and Karp, we can model this as a flow algorithm:

Formally, given a bipartite graph $G = (U \uplus W, E)$, we can construct the network $N = (V, A, c, s, t)$, where

- $V = U \uplus W \uplus \{s, t\}$
- $A = E \cup (U \times \{s\}) \cup (W \times \{t\})$
- $c(e) = 1 \ \forall e \in A$

And the maximum flow f_{max} gives us our maximum matching. We can find the edges of that matching, by taking only the edges through which our flow "flows" through. val(f_{max}) gives us the size of the maximum matching. This lemma from the script confirms our ideas:

Lemma: (3.15) Yes, the above ideas are correct dear A&W student. Well done :)

2.3.2 Menger's Theorem

Using flow, we can actually prove Menger's Theorem. We will do so for the edge-connectivity case. Here, we have to show that for given u,v, the number of edge-disjoint-paths is exacltly the minimum number of edges you have to get rid of to destroy connectivity (see Menger's Theorem). To do so, we take a graph $G = (V, E)$ and create a network $N = (V', A', c', s', t')$, where

- $V' = V \cup \{s, t\}$
- $A' = \{(x, y), (y, x) \mid \{x, y\} \in E\} \cup \{(s, u), (v, t)\}$

- $c'(e) = 1 \ \forall e \in A' \setminus \{(s,u),(v,t)\}$
- $c'(s,u) = n$ (could also be tighter, for example $c'(s,u) = \deg_{out}(u)$)
- $c'(v,t) = n$

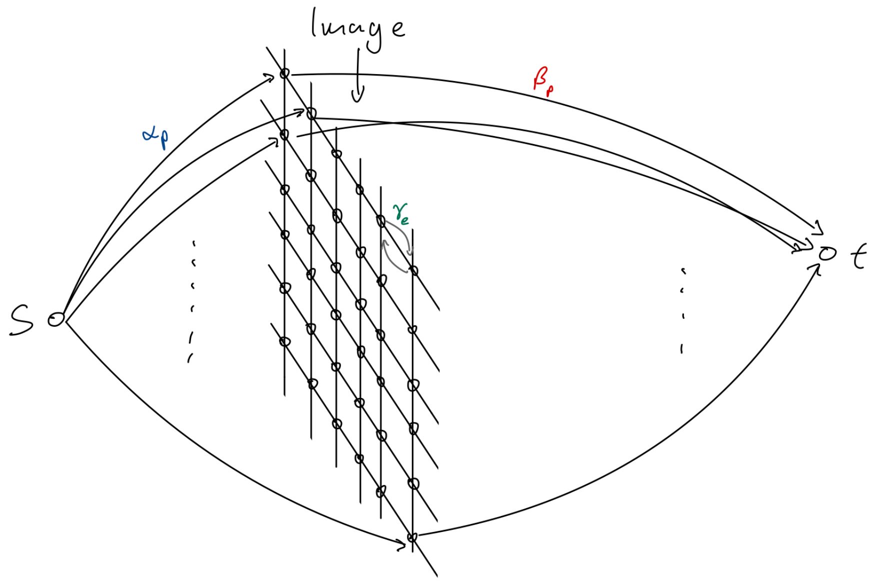


Figure 3: What is happening on a schematic level in our construction in image segmentation

2.3.3 Image Segmentation

Another application of flow is image segmentation. We want to take a grid of pixels P and separate the pixels into one group which belongs to the foreground and one group which belongs to the background.
We will model this situation as a graph $G = (P, E)$, where the edges are such that they only connect adjacent pixels. Like true mathematicians, we will make this process unnecessarily formal and introduce for each pixel p two numbers:

$$\alpha_p = \text{how likely is it that } p \text{ belongs to the foreground}$$
$$\beta_p = \text{how likely is it that } p \text{ belongs to the background}$$

Then, the set of pixels in the foreground would be A and the set of pixels in the background would be B as follows:

$$A := \{p \in P \mid \alpha_p > \beta_p\}$$
$$B := \{p \in P \mid \beta_p \leq \alpha_p\}$$

We would still like some sort of smoothing to prevent too fine grained results. To that end, we will introduce another parameter but this time for the edges

$$\gamma_{\{p,q\}\in E} = \text{penalty for putting pixels p and q in different sets}$$

with which we will penalize separation of adjacent pixels. In total, what we want to do is apply this separation while optimizing (minimizing) the penalty, which can be the following values

1. $\sum_{p \in A} \beta_p$ (background-values of pixels which are in the foreground)
2. $\sum_{p \in B} \alpha_p$ (foreground-values of pixels which are in the background)

3. $\sum_{e\in E, |e\cap A|=1} \gamma_e$

So in total, our goal is to minimize our total penalty

$$q'(A,B) := \sum_{p\in A} \beta_p + \sum_{p\in B} \alpha_p + \sum_{e\in E, |e\cap A|=1} \gamma_e$$

Within some sort of network by finding the minimum cut, which will hopefully be our penalty q'. Our idea is now to construct the network as follows: Let the set of vertices be P with the addition of another source s and target t. We will add an edge from s to every pixel with capacity α_p and an edge from every pixel to t with capacity β_p. Each edge between two pixels p, q we will replace with two edges going in both directions with capacity $\gamma_{\{p,q\}}$ After calculating the maximum flow, we can take the corresponding minimum cut (S,T) and set $A = S \setminus \{s\}$ and $B = T \setminus \{t\}$.