1 Network Flows

This is an excerpt from the book Průvodce labyrintem algoritmů by Martin Mareš and Tomáš Valla, translated to English by its first author.

1.1 Preliminaries

We will use the following notation:

- V and E are the sets of *vertices* and directed *edges* of the network.
- $n = |V|$ denotes the number of vertices, $m = |E|$ the number of edges.
- $s, t \in V$ are the *source* and the *target* respectively.
- uv will denote a directed edge (arc) from u to v.
- Each edge uv is assigned a non-negative *capacity* $c(uv)$ and non-negative *flow* $f(uv)$.
- $f^+(v) = \sum_{u:uv \in E} f(uv)$ is the total *inflow* of a vertex v.
- $f^-(v) = \sum_{w:vw \in E} f(vw)$ is the total *outflow* of a vertex v.
- $f^\Delta(v) = f^+(v) - f^-(v)$ is the *excess* of a vertex v. In a proper flow, it is zero everywhere except the source and the target. (This is called the *Kirchhoff's law.*)
- $r(uv) = c(uv) - f(uv) + f(vu)$ is the *residual capacity* of an edge uv. It specifies how much can be sent from u to v by adding to $f(uv)$ and/or subtracting from $f(vu)$.
- Edges with zero residual capacity are called *saturated.* A path is *non-saturated* if all its edges are non-saturated.
- For $A, B \subseteq V$, $E(A, B)$ is the set of all edges going from a vertex in A to a vertex in B. That is, $E(A, B) = E \cap (A \times B)$.

1.2 Goldberg's Algorithm

Let us introduce one more algorithm for finding the maximum flow in a network. It was discovered by Andrew Goldberg. While it is much simpler than the Dinitz's algorithm, it will match its speed and even beat it after slight improvements. We will however pay for the simplicity by effort needed to analyze the algorithm and prove its properties.

Preflows, excesses, and heights

Traditional maximum flow algorithms start with an everywhere-zero flow and they gradually improve it until it becomes maximum. Goldberg's algorithm does the opposite: it starts with edge values which are not even a flow. Then it gradually redistributes the values until they become a valid flow and indeed a maximum one.

Definition: A function $f : E \to \mathbb{R}_0^+$ is a *preflow* in the network (V, E, s, t, c) if it satisfies both of the following conditions:

- $\forall e \in E : f(e) \leq c(e)$ (edge capacities are not exeeded),
- $\forall v \in V \setminus \{s, t\} : f^\Delta(v) \geq 0$ (excesses of vertices are non-negative).

We see that each flow is a preflow, but not vice versa. The algorithm will always maintain a preflow. It will try to clear out excess of all vertices except for the source and the target. The primary tool for that will be pushing of excess:

Definition: *Pushing the excess* through an edge uv is allowed if $f^\Delta(u) > 0$ and $r(uv) > 0$. It is performed by sending $\delta = \min(f^\Delta(u), r(uv))$ units of flow through uv – similary to previous algorithms by either adding in the direction of the edge or subtracting against its direction.

Observation: A push will change excesses and residual capacities as follows:

$$\begin{aligned} f'^\Delta(u) &= f^\Delta(u) - \delta \\ f'^\Delta(v) &= f^\Delta(v) + \delta \\ r'(uv) &= r(uv) - \delta \\ r'(vu) &= r(vu) + \delta \end{aligned}$$

We would like to iterate pushing until we transport all excess to the target or possibly back to the source. We must do it carefully to avoid infinite loops, though. Each vertex will be assigned its *height* – a non-negative integer $h(v)$.

We will push only „downhill" – from a higher vertex to a lower one. In case we happen to find a vertex with non-zero excess and no downhill non-saturated edges, we *raise* the vertex by incrementing its height by 1. Raising will be repeated until the vertex is high enough to admit a push.

This leads to the following algorithm:

Algorithm GOLDBERG
Input: A network (V, E, s, t, c)

1. Initialize heights: ◁ *source at height n, other vertices at height 0*
2. $h(s) \leftarrow n$
3. $h(v) \leftarrow 0$ for all $v \neq s$
4. Initialize the pre-flow: ◁ *all edges from s at maximum*
5. $f \leftarrow$ everywhere zero function
6. $f(sv) \leftarrow c(sv)$ whenever $sv \in E$

7. While there is a vertex $u \neq s, t$ such that $f^\Delta(u) > 0$:
8. If there exists an edge uv with $r(uv) > 0$ and $h(u) > h(v)$, *push excess* through uv.
9. Otherwise *raise* u: $h(u) \leftarrow h(u) + 1$.

Output: Maximum flow f

Algorithm analysis*

The algorithm is simplicity itself, but it is hardly clear that it does anything useful. We will gradually prove several invariants and lemmata, which will finally lead to a proof of correctness and time complexity.

Invariant A (basic): In every step of the algorithm:

1. The function f is a preflow.
2. The height $h(v)$ of any vertex v is non-decreasing.
3. $h(s) = n$ and $h(t) = 0$.
4. $f^\Delta(v) \geq 0$ at every vertex $v \neq s$.

Proof: By induction on the number of iterations (steps 7.–9. of the algorithm):

- The invariant holds after initialization: excess of every vertex besides the source is non-negative, heights are also as required.
- Upon a push: By definition of a push, capacities are never exceeded and no negative excess is created. Heights do not change.
- Upon a raise: Only heights change, but only at vertices different from the source and the target. Also, they always increase. □

Invariant D (on drop): There is no edge uv with positive residual capacity and *drop* $h(u) - h(v)$ greater than 1.

Proof: By induction on the computation steps. At the beginning, all edges from the source have zero residual capacity and all other edges have drop 0 or they go uphill (having negative drop). Later on, there are only two possibilities how the invariant could be broken:

- By raising a vertex u with an outgoing edge uv of positive residual capacity and drop 1. This does not happen, because the algorithm would prefer a push over the raise.
- By increasing residual capacity of an edge with drop greater than 1. This is also impossible, because residual capacity is increased only when we push against the direction of the edge, but we never push uphill. □

Lemma C (on correctness): If the algorithm stops, f is a maximum flow.

Proof: We first prove than f satisfies the conditions of a flow: It is bounded by capacities (this condition is the same for flows as for pre-flows), so it suffices to check the Kirchhoff's law. It requires all excesses to be zero except at the source and at the sink. It must indeed hold, because a non-zero excess at a regular vertex would keep the algorithm running.

Let us prove that f is indeed maximum. For sake of contradiction, let us assume that it is not. From the correctness of Ford-Fulkerson algorithm, it follows that there must exist a non-saturated path from the source to the target. We pick one such path. The source is still at height n, the target still at 0 (see invariant **A**). The total drop of this path is therefore n, but since the path contains at most $n-1$ edges, at least one edge must have drop greater than 1. This contradicts invariant **D**. □

Invariant P (path to source): Let v be a vertex with positive excess $f^\Delta(v)$. Then there exists a non-saturated path from v to the source.

Proof: Let us consider the set

$$A := \{u \in V \mid \text{there is a non-saturated path from } v \text{ to } u\}.$$

We will proceed to show that this set contains the source.

We will replay a standard trick: we consider the sum of excesses of all vertices in A, which is a certain linear combination of flows on edges. The contribution of all edges lying entirely inside A or entirely outside it will contribute will be zero. The contributions of the remaining edges, which have exactly one endpoint in A, yields:

$$\sum_{u\in A} f^\Delta(u) = \underbrace{\sum_{ba\in E(V\setminus A,A)} f(ba)}_{=0} - \underbrace{\sum_{ab\in E(A,V\setminus A)} f(ab)}_{\geq 0} \leq 0.$$

We will show that the first brace is equal to zero (let us follow figure 1.1). Consider an edge ab ($a \in A$, $b \in V \setminus A$). Its residual capacity must be zero — otherwise b would belong to A — so the flow through ab must be zero.

The second brace is obviously non-negative, because it is a sum of non-negative terms.

Therefore the sum of excesses over the set A is non-positive. However, A contains at least one vertex with positive excess, namely v. So A must also contain at least one vertex with negative excess, but the only such vertex is the source s. This concludes that s lies in A, so there is a non-saturated path from v to s. □

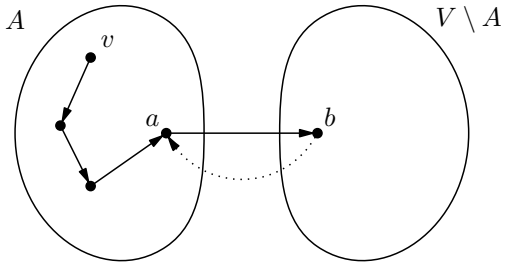

Figure 1.1: Situation in the proof of invariant **C**

Invariant H (on height): For each vertex v, we have $h(v) \leq 2n$.

Proof: If there happens to be a vertex v of height $h(v) > 2n$, consider the situation when it was last raised. Its height before the raise was at least $2n$ and it had positive excess. According to invariant **C**, there was a non-saturated path from v to the source. This path has total drop at least n, but at most $n-1$ edges, so at least one edge has drop 2 or more. Again, this contradicts invariant **D**. □

Lemma R (number of raises): There are at most $2n^2$ raises in every computation.

Proof: By the previous invariant, every of the n vertices can be raised at most $2n$ times. □

It remains to determine the maximum number of pushes. For this purpose, we will distinguish between two kinds of pushes:

Definition: A push through an edge uv is *saturating*, if it causes the residual capacity $r(uv)$ to drop to zero. Otherwise, we consider it *non-saturating* and we notice that the excess $f^{\Delta}(u)$ becoems zero (this can happen in some saturating pushes, too).

Lemma S (saturating pushes): There are at most nm saturating pushes.

Proof: Let us fix an edge uv and count saturating pushes through it.

After the first saturating push from u to v, the residual capacity of uv drops to zero. At this moment, u has to be higher than v (by invariant **D**, the height difference is exactly 1). Before the next push through the same edge, the residual capacity must be increased again. This can happen only by a push in the opposite direction, but this requires v to rise above u first. After $r(uv)$ becomes non-zero, u must be again raised above v, which requires at least 2 raises (one to the level of v, one above v).

We have shown that there are at least two raises of u between every two consecutive saturating pushes through uv. We know from invariant **H** that u is raised at most $2n$

times, so there are at most n saturating pushes through uv. Summing over all edges, we get the claimed bound nm. □

We have proved the previous two lemmata in a ''local'' way — raises were counted for each vertex separately, saturating pushes for each edge. This approach does not seem to work for non-saturating pushes, because too many of them can happen through a single edge. It is still possible to prove a nice bound on the total number of such pushes.

We will use a potential-based technique. We will define a *potential,* which is a certain *non-negative* function assigned to the current state of the computation. We will consider the effect of each operation on the potential. We will show that the total number of operations which decrease the potential cannot exceed much the total number of operations which increase it — otherwise, the potential would become negative at some moment.

Sometimes, the potential is straight-forward, but the next lemma requires a slightly more involved one. The guiding principle behind its choice is to let operations, whose counts we already bounded (raises and saturating pushes), contribute small positive amounts. Non-saturating pushes will always decrease the potential.

Lemma N (non-saturating pushes): There are $\mathcal{O}(n^2m)$ non-saturating pushes.

Proof: Let us consider the following potential:

$$\Phi := \sum_{\substack{v \neq s,t \\ f^{\Delta}(v)>0}} h(v).$$

In other words, each vertex with a positive excess contributes its height. The potential evolves during the computation as follows:

- At the beginning, we have $\Phi = 0$.
- We always have $\Phi \geq 0$, because the potential is a sum of non-negative heights.
- A raise operation increases Φ by 1. (For a vertex to be raised, its excess must be positive, so it already contributed to the sum. Now, it contributes one more.) As we know that the total number of raises is at most $2n^2$, all raises together increase the potential by at most $2n^2$.
- A saturating push through an edge uv increases Φ by at most $2n$: If v had zero excess, the excess becomes positive, so v will start contributing to Φ by $h(v) \leq 2n$. The vertex u already contributed and it will either continue doing so, or its excess drops to zero and the potential decreases. By lemma **S**, at most nm saturating pushes happen, so they increase Φ by at most $2n^2m$ altogether.

- Finally, when we do a non-saturating push through uv, the potential decreases by the height of u (the excess of u certainly became zero) and possible increases by the height of v (if v had zero excess beforehands). As $h(v) = h(u) - 1$, the potential decreased by at least 1.

The total increase of Φ is therefore at most $2n^2 + 2n^2m = \mathcal{O}(n^2m)$. Each non-saturating push causes it to decrease, so the total number of non-saturating pushes cannot exceed $\mathcal{O}(n^2m)$. □

Implementation

It remains to find a representation of the network and vertex heights, so that we can find vertices with excess and non-saturated downhill edges efficiently.

We will keep a list X of all vertices with positive excess. Every time we change the excess of a vertex, we can update this list in constant time — either we add the vertex to the list, or we remove it. (It will be useful to let each vertex remember a pointer to its position in the list.) This allows us to find a vertex with excess in constant time.

For every vertex u, we will also keep a list $L(u)$. It will contain all non-saturated edges which go downhill from u (their drop is 1). Again, we can update these lists in constant time upon every change of residual capacity.

We will reach the following time complexities of operations:

- *Initialization* of the algorithm — $\mathcal{O}(m)$ trivially.
- *Selection of a vertex* with positive excess and finding a downhill non-saturated edge leading from it — $\mathcal{O}(1)$ by looking at the heads of the particular lists.
- *Pushing excess* through an edge uv — changes of residual capacities $r(uv)$ and $r(vu)$ cause updates of lists $L(u)$ and $L(v)$, changes of excesses $f^\Delta(u)$ and $f^\Delta(v)$ cause updates of the list X. All of this happens in time $\mathcal{O}(1)$.
- *Raise of a vertex* u can cause a flat (drop 0) non-saturated edge to become downhill. Also, a downhill edge can become flat. Thus we have to enumerate all incoming and outgoing edges of u and update the lists $L(u)$ and $L(v)$. Since there are up to $2n$ such edges, the updates take time $\mathcal{O}(n)$.

We see that raises are expensive, but they are also relatively rare. On the contrary, pushes are frequent, so we are glad they take constant time.

Theorem: The Goldberg's algorithm finds a maximum flow in time $\mathcal{O}(n^2m)$.

Proof: Initialization takes $\mathcal{O}(m)$ time. Then the algorithm performs at most $2n^2$ raises (by lemma **R**), at most nm saturating pushes (lemma **S**) and $\mathcal{O}(n^2m)$ non-saturating

pushes (lemma **N**). Multiplying by the complexities of operations yields total time $\mathcal{O}(n^3 + nm + n^2m) = \mathcal{O}(n^2m)$. Lemma **C** guarantees that when the algorithm stops, it outputs maximum flow. □

Exercises

1. Analyse behavior of Goldberg's algorithm on networks with unit capacities. Will it be faster than the other algorithms?

2. What would happen if we initialized the height of the source to $n-1$, $n-2$, or perhaps $n-3$?

1.3* Improved Goldberg's algorithm

The described version of Goldberg's algorithm reached the same time complexity as Dinitz's algorithm. Surprisingly, a minor change in the algorithm leads to a substantial speedup. It suffices to always select the highest vertex of those with excess.

The dominant term in the complexity of the original algorithm was $\mathcal{O}(n^2m)$ due to non-saturating pushes. We will therefore prove a stronger bound for the modified algorithm.

Lemma N': Goldberg's algorithm with selection of the highest vertex performs $\mathcal{O}(n^3)$ non-saturating pushes.

Proof: We will employ the potential method again. We split vertices to levels by their height. We consider the *topmost level with excess:*

$$H := \max\{h(v) \mid v \neq s,t \ \wedge \ f^{\Delta}(v) > 0\}.$$

We divide the computation to *phases.* A phase ends by a change of H. Either H increases, which means that some vertex with excess at level H was raised, so the increase was by 1. Or H decreases, which happens when the last excess at level H was drained by a push. In such cases, the destination of the push must lie at level $H-1$, so H will decrease by 1 exactly.

As we know that there are $\mathcal{O}(n^2)$ raises, H can increase $\mathcal{O}(n^2)$ times. However, H is always non-negative, so the total number of its decreases must be also $\mathcal{O}(n^2)$. Therefore, we have $\mathcal{O}(n^2)$ phases in total.

During a single phase, there can be at most one non-saturating push from each vertex. This is because a non-saturating push through an edge uv clears out the excess of u. Before the next push from u, this excess must become positive again. This must happen

by a subsequent push to u, but during the current phase, there no vertices higher than u with excess, so the phase must end first.

This implies that the number of non-saturating pushes per phase is at most n. In all $\mathcal{O}(n^2)$ phases, it is $\mathcal{O}(n^3)$ pushes. □

In fact, this bound is not tight. A more complex potential argument leads to a stronger bound, which is especially useful for sparse graphs. We will state it without proof.

Lemma N'': There are $\mathcal{O}(n^2\sqrt{m})$ non-saturating pushes.

Exercises

1. Design an implementation of the improved Goldberg's algorithm with selection of the highest vertex with excess. Try to reach total time complexity $\mathcal{O}(n^2\sqrt{m})$.