Lemma 72

The number of non-saturating pushes performed is at most $\mathcal{O}(n^2 m)$.

Proof.

- Define a potential function $\Phi(f) = \sum_{\text{active nodes}\, v} \ell(v)$
- A saturating push increases Φ by at most $2n$.
- A relabel increases Φ by at most 1.
- A non-saturating push decreases Φ by at least 1 as the node that is pushed from becomes inactive and has a label that is strictly larger than the target.
- Hence,

$$\begin{aligned}\#\text{non-saturating\_pushes} &\le \#\text{relabels} + 2n \cdot \#\text{saturating\_pushes} \\ &\le \mathcal{O}(n^2 m)\ .\end{aligned}$$

Analysis

There is an implementation of the generic push relabel algorithm with running time $\mathcal{O}(n^2 m)$.

For every node maintain a list of admissable edges starting at that node. Further maintain a list of active nodes.

A push along an edge (u, v) can be performed in constant time

- check whether edge (v, u) needs to be added to G_f
- check whether (u, v) needs to be deleted (saturating push)
- check whether u becomes inactive and has to be deleted from the set of active nodes

A relabel at a node u can be performed in time $\mathcal{O}(n)$

- check for all outgoing edges if they become admissable
- check for all incoming edges if they become non-admissable

13.2 Relabel to front

For special variants of push relabel algorithms we organize the neighbours of a node into a linked list (possible neighbours in the residual graph G_f). Then we use the discharge-operation:

Algorithm 48 discharge(u)

```
1: while u is active do
2:     v ← u.current-neighbour
3:     if v = null then
4:         relabel(u)
5:         u.current-neighbour ← u.neighbour-list-head
6:     else
7:         if (u, v) admissable then push(u, v)
8:         else u.current-neighbour ← v.next-in-list
```


13.2 Relabel to front

Lemma 73

If $v = \mathrm{null}$ in line 3, then there is no outgoing admissable edge from u.

The lemma holds because push- and relabel-operations on nodes different from u cannot make edges outgoing from u admissable.

This shows that discharge(u) is correct, and that we can perform a relabel in line 4.

13.2 Relabel to front

Algorithm 49 relabel-to-front(G, s, t)

```
1:  initialize preflow
2:  initialize node list L containing V \ {s,t} in any order
3:  foreach u ∈ V \ {s,t} do
4:      u.current-neighbour ← u.neighbour-list-head
5:  u ← L.head
6:  while u ≠ null do
7:      old-height ← ℓ(u)
8:      discharge(u)
9:      if ℓ(u) > old-height then
10:         move u to the front of L
11:     u ← u.next
```


13.2 Relabel to front

Lemma 74 (Invariant)

In Line 6 of the relabel-to-front algorithm the following invariant holds.

1. *The sequence L is topologically sorted w.r.t. the set of admissable edges; this means for an admissable edge (x, y) the node x appears before y in sequence L.*
2. *No node before u in the list L is active.*

Proof:

- Initialization:
  1. In the beginning s has label $n \geq 2$, and all other nodes have label 0. Hence, no edge is admissable, which means that any ordering L is permitted.
  2. We start with u being the head of the list; hence no node before u can be active
- Maintenance:
  1. - Pushes do no create any new admissable edges. Therefore, not relabeling u leaves L topologically sorted.
     - After relabeling, u cannot have admissable incoming edges as such an edge (x, u) would have had a difference $\ell(x) - \ell(u) \geq 2$ before the re-labeling (such edges do not exist in the residual graph).

     Hence, moving u to the front does not violate the sorting property for any edge; however it fixes this property for all admissable edges leaving u that were generated by the relabeling.

13.2 Relabel to front

Proof:

- Maintenance:
  2. If we do a relabel there is nothing to prove because the only node before u' (u in the next iteration) will be the current u; the discharge(u) operation only terminates when u is not active anymore.

     For the case that we do a relabel, observe that the only way a predecessor could be active is that we push flow to it via an admissable arc. However, all admissable arc point to successors of u.

Note that the invariant for $u = \text{null}$ means that we have a preflow with a valid labelling that does not have active nodes. This means we have a maximum flow.

13.2 Relabel to front

Lemma 75

There are at most $\mathcal{O}(n^3)$ calls to discharge(u).

Every discharge operation without a relabel advances u (the current node within list L). Hence, if we have n discharge operations without a relabel we have $u = \text{null}$ and the algorithm terminates.

Therefore, the number of calls to discharge is at most $n(\#\mathit{relabels} + 1) = \mathcal{O}(n^3)$.

13.2 Relabel to front

Lemma 76

The cost for all relabel-operations is only $\mathcal{O}(n^2)$.

A relabel-operation at a node is constant time (increasing the label and resetting $u.\mathit{current\text{-}neighbour}$). In total we have $\mathcal{O}(n^2)$ relabel-operations.

13.2 Relabel to front

Note that by definition a saturing push operation $(\min\{c_f(e), f(u)\} = c_f(e))$ can at the same time be a non-saturating push operation $(\min\{c_f(e), f(u)\} = f(u))$.

Lemma 77

*The cost for all saturating push-operations that are **not** also non-saturating push-operations is only $\mathcal{O}(mn)$.*

Note that such a push-operation leaves the node u active but makes the edge e disappear from the residual graph. Therefore the push-operation is immediately followed by an increase of the pointer $u.\mathit{current\text{-}neighbour}$.

This pointer can traverse the neighbour-list at most $\mathcal{O}(n)$ times (upper bound on number of relabels) and the neighbour-list has only $\mathit{degree}(u) + 1$ many entries ($+1$ for null-entry).

13.2 Relabel to front

Lemma 78

The cost for all non-saturating push-operations is only $\mathcal{O}(n^3)$.

A non-saturating push-operation takes constant time and ends the current call to discharge(). Hence, there are only $\mathcal{O}(n^3)$ such operations.

Theorem 79

The push-relabel algorithm with the rule relabel-to-front takes time $\mathcal{O}(n^3)$.