Push–Relabel Algorithms and Relabel-to-Front

CS5800 — Lecture Notes (written with Claude AI)

August 5, 2026

Sources: CLRS 3rd ed., §26.4–26.5; worked example adapted from a ChatGPT walkthrough, with the discharge mechanics re-derived and corrected below.

---

0. Roadmap

1. Why push–relabel is a different style of max-flow algorithm than Ford–Fulkerson.
2. Preflow, excess, height functions — the core bookkeeping.
3. The two local operations: `Push` and `Relabel`.
4. **`Discharge(u)`: the operation the main algorithm actually calls** — pick an active vertex, then push/relabel *it alone*, repeatedly, until its excess is gone.
5. Admissible edges — the structure that makes a smart discharge *order* possible.
6. **Relabel-to-front**: discharge vertices in a self-adjusting list order, achieving $O(V^3)$.
7. Full worked example, step by step.
8. Why it's correct, why it's fast.

---

1. The big idea: preflow instead of flow

Ford–Fulkerson finds augmenting *paths* from s to t one at a time — a global operation that looks at the whole residual graph. Push–relabel instead works **locally**: it repeatedly looks at *one* vertex and its neighbors.

To make this possible, we relax the flow-conservation constraint. Instead of a flow, we maintain a **preflow** f: capacities are still respected, but a vertex is now allowed to receive more than it sends out. Define the **excess** at u:

$$e(u) = \sum_v f(v,u) - \sum_v f(u,v) \ \geq 0.$$

A vertex $u \in V - \{s, t\}$ with $e(u) > 0$ is called **overflowing** (or *active*). s and t are never considered overflowing.

Fluid intuition (CLRS). Think of the graph as pipes (edges) of fixed capacity connecting junctions (vertices), each junction sitting atop a reservoir that can hold arbitrary excess fluid, and each junction's platform having a **height**. Fluid can only be pushed *downhill* — from a higher vertex to a lower one. We fix $h(s) = |V|$ (source is highest) and $h(t) = 0$ (sink is lowest); all other heights start at 0 and only ever increase. The algorithm:

- sends flow downhill from s, filling outgoing pipes to capacity;
- whenever a vertex has excess but every neighbor reachable by a pipe is at the same height or higher (nowhere downhill to push), it **raises its own height** just enough to open a downhill pipe — this is called *relabeling*;
- repeats until no vertex overflows.

Once nothing overflows, the preflow is automatically a maximum flow.

---

2. Height functions

A function $h : V \to \mathbb{N}$ is a **height function** for preflow f if:

- $h(s) = |V|$, $h(t) = 0$, and
- for every **residual** edge $(u, v) \in E_f$ (i.e. $c_f(u, v) > 0$): $h(u) \leq h(v) + 1$.

  Lemma 26.12. If $h(u) > h(v) + 1$, then (u, v) cannot be a residual edge.

This is the load-bearing fact of the whole method: *height differences bound where flow can possibly go.* Once we show s–t has no residual path (via heights), the max-flow/min-cut theorem finishes the correctness proof for us — no need to reconstruct an explicit cut.

---

3. The two basic operations

Applicability conditions matter as much as the actions — an operation is illegal outside its stated conditions.

Push(u, v)

> **Applies when:** u is overflowing, $c_f(u, v) > 0$, and $h(u) = h(v) + 1$ (this is what makes edge (u, v) **admissible**). **Action:** push $\delta = \min(e(u), c_f(u, v))$ units of flow from u to v; update f, then $e(u)$ $-= \delta$, $e(v)$ $+= \delta$.

- **Saturating push**: $\delta = c_f(u, v)$ — the edge becomes full and *disappears* from the residual graph.
- **Non-saturating push**: $\delta = e(u)$ — u's excess drops to exactly 0 (Lemma 26.13); u stops overflowing.

Flow always moves strictly downhill, one level at a time, never more.

Relabel(u)

> **Applies when:** u is overflowing and **no** admissible edge leaves u (i.e. for every residual edge (u, v), $h(u) \leq h(v)$). **Action:** $h(u) \leftarrow 1 + \min\{\, h(v) : (u, v) \in E_f \,\}$.

Relabeling always applies because u is overflowing, hence has *some* residual outgoing edge; the min above is over a nonempty set. After relabeling, u gains at least one admissible outgoing edge and *none* newly enters it.

> **Key emphasis:** heights are **monotone non-decreasing**, and every `Relabel` call strictly *increases* $u.h$ by at least 1 (Lemma 26.15). This single fact drives essentially every bound in this lecture.

---

4. `Discharge(u)`: the operation the algorithm actually runs

Everything above described two *primitive* moves. In practice, the algorithm never calls them in isolation — it commits to a vertex and doesn't let go until that vertex is happy. **Once we pick up an overflowing vertex u, we push and relabel it, and *only* it, over and over, until $u.e = 0$.** That whole unit of work is called discharging u.

Each vertex u keeps a **neighbor list** $u.N$ (everyone u has an edge to or from) and a pointer u.current into that list, so that repeated scans pick up where they left off instead of restarting from the head every time.

```
DISCHARGE(u)
1  while u.e > 0
2      v = u.current
3      if v == NIL
4          RELABEL(u)
5          u.current = u.N.head
6      elif c_f(u,v) > 0 and u.h == v.h + 1
7          PUSH(u,v)
8      else
9          u.current = v.next-neighbor
```

Each iteration does exactly one of three things:

1. **Ran off the end of the list** ($v = \text{NIL}$) $\Rightarrow$ relabel u, reset the pointer to the head. (A full pass through $u.N$ without draining u means every edge left u was inadmissible — precisely the condition that licenses `Relabel`.)
2. **Current neighbor is admissible** $\Rightarrow$ push.
3. **Current neighbor is inadmissible** $\Rightarrow$ advance the pointer, nothing else changes.

`DISCHARGE(u)` always ends with a push — the loop only exits when $u.e = 0$, and neither relabeling nor advancing the pointer touches $u.e$.

The main algorithm

> **while** some vertex in $V - \{s, t\}$ is overflowing: pick one such vertex u and call `DISCHARGE(u)`.

That's the whole algorithm, at the granularity that matters for this lecture. Two things are worth separating cleanly:

- **Correctness never depends on which active vertex we pick next.** Every step `DISCHARGE` performs is a legal `Push` or `Relabel` in isolation, so the algorithm halts only when nothing overflows — at that point the preflow is an honest flow, and (via Lemma 26.12's height argument, §2) it is automatically maximum. We do not need to think about arbitrary single-operation interleavings to see this; committing to fully discharging one vertex at a time is just one particular — and particularly convenient — legal schedule.
- **Running time entirely depends on which active vertex we pick next.** A careless choice of order can force a large number of expensive relabels and pushes. The rest of this lecture is about one specific, provably efficient choice of order: **relabel-to-front**, which discharges vertices in a self-reorganizing list and achieves $O(V^3)$.

---

5. Admissible edges — the structure relabel-to-front exploits

Fix a preflow f and height function h. Edge (u, v) is **admissible** if $c_f(u, v) > 0$ and $h(u) = h(v) + 1$; call $G_{f,h} = (V, E_{f,h})$ the **admissible network**.

> **Lemma 26.26.** $G_{f,h}$ is a DAG.
>
> *Proof idea:* along any cycle, heights would have to strictly decrease all the way around and yet return to their starting value — impossible.
>
> **Lemma 26.27.** `Push` never creates new admissible edges (it may destroy the one it uses, if saturating).
>
> **Lemma 26.28.** `Relabel(u)` creates at least one new admissible edge *leaving* u, and none *entering* u.

Put together: **relabeling is the only operation that changes the admissible network, and it always does so in a way compatible with moving u to the "front" of a topological order.** This is the entire mechanism relabel-to-front is built on — we are about to maintain a vertex list L that stays a topological sort of $G_{f,h}$ at all times.

---

6. The relabel-to-front algorithm

We maintain a doubly linked list L of $V - \{s, t\}$. **The order of discharges is: walk down L from the front, discharging each vertex in turn; whenever discharging a vertex raises its height, move it to the front of L before continuing.**

```
RELABEL-TO-FRONT(G, s, t)
1  INITIALIZE-PREFLOW(G, s)
2  L = V - {s, t}, in any order
3  for each u in V - {s,t}:  u.current = u.N.head
4  u = L.head
5  while u != NIL:
6      old-height = u.h
7      DISCHARGE(u)
8      if u.h > old-height:
9          move u to the front of L
10     u = u.next
```

Reading line 10 correctly is essential. Line 9 (if taken) physically splices u out of its old spot and re-inserts it at the head of L — and line 10 then reads `u.next` *after* that splice, i.e. in u's new position. Since u is now the head, `u.next` is whichever vertex u just displaced: **the vertex that used to be at the front of the list**, not the vertex that used to follow u before it moved.

Concretely: if $L = [a, b, c, d, e]$ and discharging b raises its height, L becomes $[b, a, c, d, e]$, and the *next* vertex discharged is a — the old head — not c. This is what makes the heuristic actually do something: a freshly relabeled vertex gets reconsidered immediately, before the scan is allowed to move on to fresh territory.

Why "move to front" is the right heuristic

- Heights only increase, and admissible edges only run downhill by exactly 1.
- A vertex that just got relabeled has a *fresh* set of admissible edges and can likely push again immediately — so we revisit it right away instead of waiting a full lap of L.
- Moving it to the front keeps L a **topological sort of the admissible network** — every vertex before u in L has no admissible path *to* u. That is exactly what makes a single left-to-right sweep of L sufficient to drain everything, without excess ever "sneaking backward" past vertices already finished.
- The list self-organizes toward non-increasing height, which is what the running-time proof in §9 formalizes.

---

7. Worked example

Graph. $V = \{s, a, b, c, d, e, f, t\}$, $|V| = 8$. Two branches out of s:

Edge	Cap	Edge	Cap
$s \to a$	10	$s \to b$	9
$a \to c$	4	$b \to d$	5
$c \to e$	4	$b \to f$	3
		$d \to f$	2
		$d \to t$	6
		$f \to t$	4

Branch $a \to c \to e$ is a dead end — e has no edge to t at all, so *nothing* pushed into this branch can ever reach the sink. Every unit that enters it must eventually flow all the way back out through c, then a, then into s itself: a genuine 3-node round trip. **Branch $b \to \{d, f\} \to t$** mostly succeeds, but b's own capacity out ($5 + 3 = 8$) is tighter than $s \to b$ (9), so 1 unit sent to b also has nowhere to go and bounces straight back to s. The true bottleneck of this whole network turns out to be b's two outgoing pipes, not either edge out of s — the max flow will be 8, not $10+9 = 19$.

Initialization. $h(s) = 8$; all other heights $= 0$. Saturate $s \to a$ ($e(a) = 10$) and $s \to b$ ($e(b) = 9$). Initial list $L = [a, b, c, d, e, f]$.

Notice the forward capacities *inside* each branch ($a \to c$: only 4 out of a's 10; $b \to d, b \to f$: only 8 out of b's 9) are deliberately smaller than the excess sitting behind them. That mismatch is what forces the double relabels below: a vertex opens one cheap admissible edge, drains part of its excess into it, and then — still overflowing, with that edge now saturated — has no choice but to relabel *again*, immediately, to find its only remaining option.

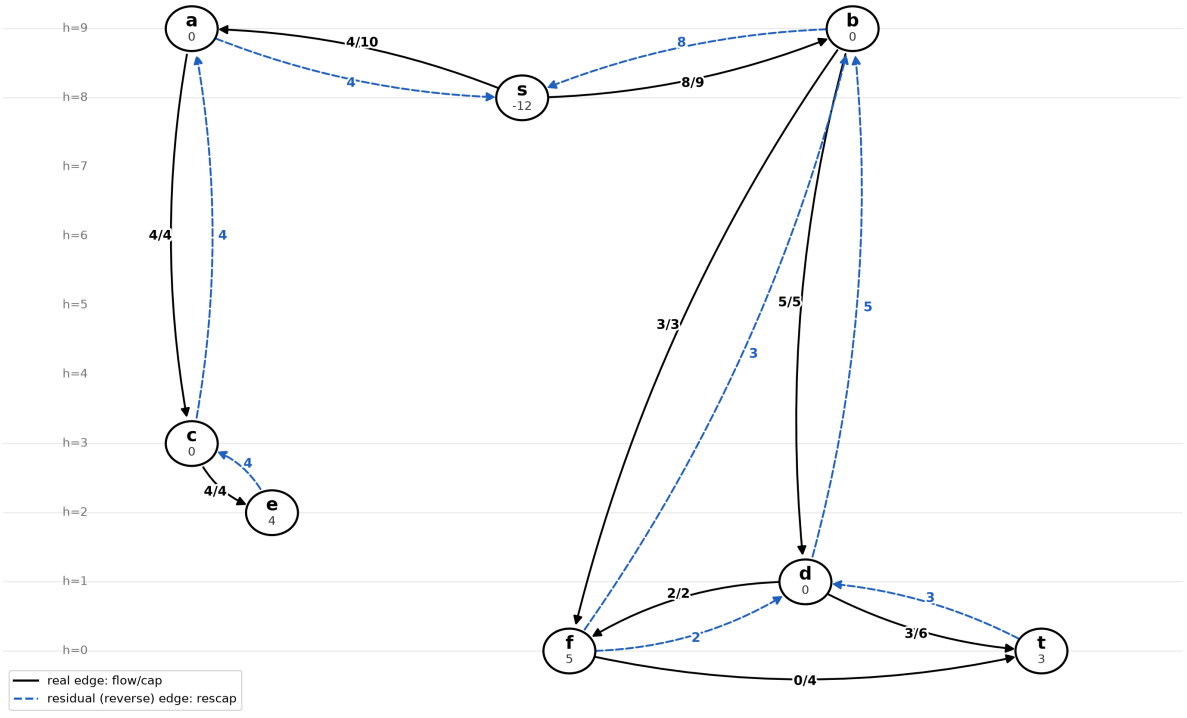
Every relabel gets its own row. A discharge with no relabel at all is a single row (a constant height, plus whatever it pushed). A discharge that needed k relabels to finish is k rows: each row is one relabel *together with the push(es) it immediately unlocked*, up to the next relabel (or the end of that discharge). The u column names the vertex only on the *first* line of its discharge — a blank u means "still the same vertex, one relabel later." $e(u)$ shows that vertex's excess *going into* this row — the amount that's driving the relabel or push(es) shown — so $e(u) = 0$ is exactly what marks a no-op. $h(u)$ always shows the height for this row (a single value, or "old→new" if this row is a relabel) — it's shown even for no-op rows, since the vertex's height doesn't change just because it had nothing to push. The L column carries two lines per row, aligned digit-under-digit: **excess** on top (labeled, e.g. a:6), and directly beneath each one, that same vertex's **current height** (unlabeled, just the number) — so you can watch heights climb in real time without re-reading labels. Shading alternates every row (since each row now has at most one relabel, by construction, each row *is* one phase boundary from §9). A genuine no-op — `DISCHARGE` called on a vertex that was already at $e(u) = 0$, so its `while` loop body never runs at all — is flagged plainly: pushes just says "skip," and L is left blank (nothing changed, so there's nothing new to show).

One more convention in the pushes column: **a double arrow (⇒) marks a push that flows backward along an edge relative to that edge's original direction** — i.e. it's undoing flow sent earlier, not extending it further. $a \Rightarrow s$:6 is water returning to the source,

but the same convention applies anywhere: $e \Rightarrow c{:}4$ is e handing flow back to c along the reverse of $c \to e$, even though c is nowhere near s. A plain arrow ($\to$) always means the push follows an edge's original direction (whether that edge is being used for the first time or refilled after a partial reversal).

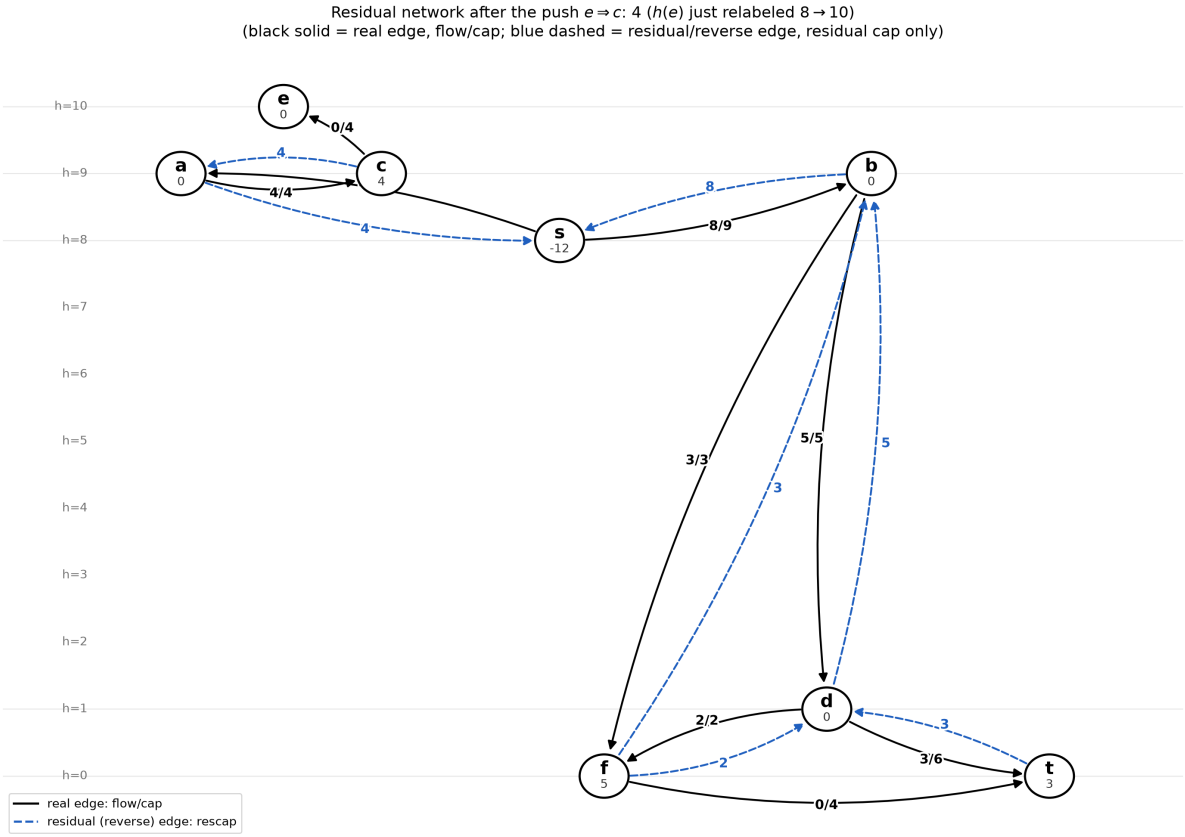
u	$e(u)$	$h(u)$	pushes (target:amount)	L: excess / height (current order)
a	10	$0 \to 1$	$a \to c{:}4^*$	$a{:}6, b{:}9, c{:}4, d{:}0, e{:}0, f{:}0$ $1,\ 0,\ 0,\ 0,\ 0,\ 0$
	6	$1 \to 9$	$a \Rightarrow s{:}6$	$a{:}0\ b{:}9\ c{:}4\ d{:}0\ e{:}0\ f{:}0$ $9\ 0\ 0\ 0\ 0\ 0$
b	9	$0 \to 1$	$b \to d{:}5^*,\ b \to f{:}3^*$	$a{:}0, b{:}1, c{:}4, d{:}5, e{:}0, f{:}3$ $9,\ 1,\ 0,\ 0,\ 0,\ 0$
	1	$1 \to 9$	$b \Rightarrow s{:}1$	$b{:}0\ a{:}0\ c{:}4\ d{:}5\ e{:}0\ f{:}3$ $9\ 9\ 0\ 0\ 0\ 0$
a	0	9	skip	
c	4	$0 \to 1$	$c \to e{:}4^*$	$c{:}0, b{:}0, a{:}0, d{:}5, e{:}4, f{:}3$ $1,\ 9,\ 9,\ 0,\ 0,\ 0$
b	0	9	skip	
a	0	9	skip	
d	5	$0 \to 1$	$d \to f{:}2^*,\ d \to t{:}3$	$d{:}0\ c{:}0\ b{:}0\ a{:}0\ e{:}4\ f{:}5$ $1\ 1\ 9\ 9\ 0\ 0$
c	0	1	skip	
b	0	9	skip	
a	0	9	skip	
e	4	$0 \to 2$	$e \Rightarrow c{:}4^*$	$e{:}0, d{:}0, c{:}4, b{:}0, a{:}0, f{:}5$ $2,\ 1,\ 1,\ 9,\ 9,\ 0$
d	0	1	skip	
c	4	$1 \to 3$	$c \to e{:}4^*$	$c{:}0\ e{:}4\ d{:}0\ b{:}0\ a{:}0\ f{:}5$ $3\ 2\ 1\ 9\ 9\ 0$

Residual network after the second push $c \to e$: 4 ($h(c)$ just relabeled $1 \to 3$)
(black solid = real edge, flow/cap; blue dashed = residual/reverse edge, residual cap only)



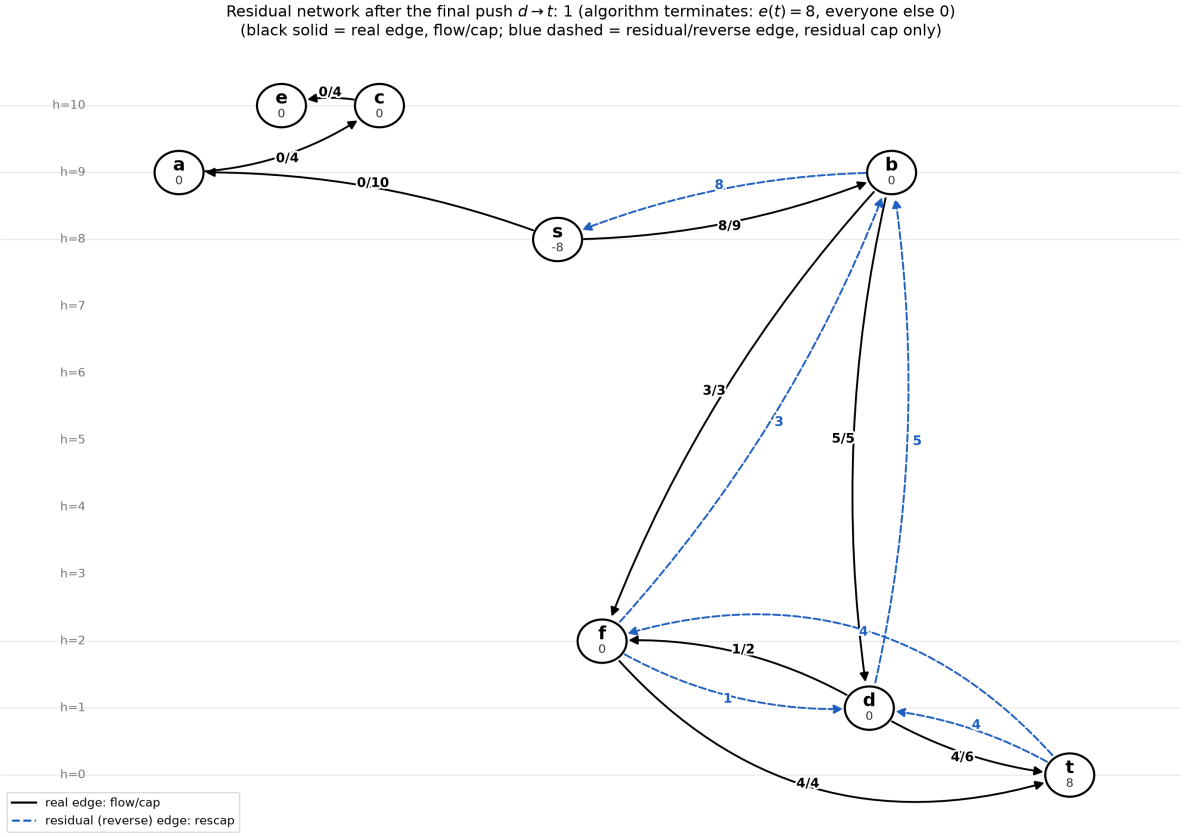
Residual network right after this push — c has just relabeled $1 \to 3$ and pushed 4 back to e again. Solid black = real edge (flow/cap); dashed blue = residual edge (residual cap only); every node is drawn at its own height, with excess printed underneath.

u	$e(u)$	$h(u)$	pushes (target:amount)	L: excess / height (current order)
e	4	$2 \to 4$	$e \Rightarrow c{:}4^*$	$e{:}0, c{:}4, d{:}0, b{:}0, a{:}0, f{:}5$ $4,\ 3,\ 1,\ 9,\ 9,\ 0$
c	4	$3 \to 5$	$c \to e{:}4^*$	$c{:}0\ e{:}4\ d{:}0\ b{:}0\ a{:}0\ f{:}5$ $5\ \ 4\ \ 1\ \ 9\ \ 9\ \ 0$
e	4	$4 \to 6$	$e \Rightarrow c{:}4^*$	$e{:}0, c{:}4, d{:}0, b{:}0, a{:}0, f{:}5$ $6,\ 5,\ 1,\ 9,\ 9,\ 0$
c	4	$5 \to 7$	$c \to e{:}4^*$	$c{:}0\ e{:}4\ d{:}0\ b{:}0\ a{:}0\ f{:}5$ $7\ \ 6\ \ 1\ \ 9\ \ 9\ \ 0$
e	4	$6 \to 8$	$e \Rightarrow c{:}4^*$	$e{:}0, c{:}4, d{:}0, b{:}0, a{:}0, f{:}5$ $8,\ 7,\ 1,\ 9,\ 9,\ 0$
c	4	$7 \to 9$	$c \to e{:}4^*$	$c{:}0\ e{:}4\ d{:}0\ b{:}0\ a{:}0\ f{:}5$ $9\ \ 8\ \ 1\ \ 9\ \ 9\ \ 0$
e	4	$8 \to 10$	$e \Rightarrow c{:}4^*$	$e{:}0, c{:}4, d{:}0, b{:}0, a{:}0, f{:}5$ $10,\ 9,\ 1,\ 9,\ 9,\ 0$



Residual network right after this push — e has just relabeled $8 \to 10$ and pushed all 4 of its excess back to c. Note that e is now the highest vertex in the whole network, higher even than a, b, and (after the next row) c.

u	$e(u)$	$h(u)$	pushes (target:amount)	L: excess / height (current order)
c	4	$9 \to 10$	$c \Rightarrow a{:}4^*$	$c{:}0\ e{:}0\ d{:}0\ b{:}0\ a{:}4\ f{:}5$ $10\ 10\ 1\ 9\ 9\ 0$
e	0	10	skip	
d	0	1	skip	
b	0	9	skip	
a	4	9	$a \Rightarrow s{:}4^*$	$c{:}0, e{:}0, d{:}0, b{:}0, a{:}0, f{:}5$ $10,\ 10,\ 1,\ 9,\ 9,\ 0$
f	5	$0 \to 1$	$f \to t{:}4^*$	$c{:}0, e{:}0, d{:}0, b{:}0, a{:}0, f{:}1$ $10,\ 10,\ 1,\ 9,\ 9,\ 1$
	1	$1 \to 2$	$f \Rightarrow d{:}1$	$f{:}0\ c{:}0\ e{:}0\ d{:}1\ b{:}0\ a{:}0$ $2\ 10\ 10\ 1\ 9\ 9$
c	0	10	skip	
e	0	10	skip	
d	1	1	$d \to t{:}1$	$f{:}0, c{:}0, e{:}0, d{:}0, b{:}0, a{:}0$ $2,\ 10,\ 10,\ 1,\ 9,\ 9$



Residual network right after this push — the final active step. d has just sent its last unit to t; every vertex now has zero excess except t. The whole a–c–e branch shows $0/\cdot$ on every edge: all 10 units that ever entered it were fully returned to s, exactly as a dead-end branch must.

u	$e(u)$	$h(u)$	pushes (target:amount)	L: excess / height (current order)
b	0	9	skip	
a	0	9	skip	

Result: $e(t) = 8$; everything else is 0. That matches the true bottleneck (b's two outgoing pipes, $5 + 3$), *not* either edge leaving s — a good reminder that the source-side cut is rarely the tight one.

What to notice, mechanically:

- **Two double-relabels right at the start.** a's first row opens the cheap edge to c and saturates it with 4 of its 10 units — but a is *still* overflowing, and its only remaining residual edge is the one straight back to s, so its second row (blank u, same vertex) is a relabel jumping from height 1 **directly to** 9 (a jump of 8, nowhere close to "+1"), pushing the leftover 6 home. b's two rows right after do the same thing, for the same reason.
- **A third, gentler double-relabel, near the end.** f's first row opens t and drains 4 of its 5 units; its second row (blank u) is a much smaller jump ($1 \to 2$) to reach d for the last 1 unit — same mechanism, tiny scale, showing this isn't only a "back-to-the-source" phenomenon.
- **Height jumps of more than 1 are common, not exotic.** Beyond the two big jumps above, look at e's very first appearance: it jumps straight from 0 to 2 in a *single* relabel (no double-relabel needed), simply because its only neighbor c was already sitting at height 1 when e first checked in.
- **The a–c–e branch returns *all* of its water to the source**, exactly as a dead-end branch must: 6 units immediately, and the remaining 4 only after a long tug-of-war — watch the height line in the last column climb in lockstep, c and e each pushing the other one level higher, until c finally clears height 10 and can reach a, which then clears it to s.
- **That tug-of-war *is* the phase structure from §9, visible.** Every row in that stretch flips shade — each one ends in a relabel, so each one *is* its own phase. Contrast that with the long unshaded/shaded runs elsewhere in the table, where nobody relabels for several rows in a row, so each such run is a single phase regardless of how many rows it spans.

---

8. Correctness of relabel-to-front

Loop invariant: at every test of line 5, L is a topological sort of the admissible network $G_{f,h}$, and no vertex *before* u in L has positive excess.

- **Initialization:** trivially true — $G_{f,h}$ has no edges yet ($h \equiv 0$ except s, t; nothing is admissible), so any order is a topological sort, and u starts at the head so nothing precedes it.
- **Maintenance:** `Push` never creates admissible edges (Lemma 26.27); `Relabel(u)` creates admissible edges only *leaving* u, never entering it (Lemma 26.28). So moving a just-relabeled u to the front keeps L topologically sorted. And since L stays topologically sorted and pushes only move flow to vertices *later* in the admissible order, excess never appears earlier in L than the current scan position.

- **Termination:** when u finally falls off the end of L (becomes NIL), the invariant says every vertex has zero excess — i.e. the preflow is a flow, and by the height/min-cut argument of §2, it must be maximum.

---

9. Running time: $O(V^3)$

Definition (phase). A *phase* is a maximal stretch of time between two consecutive `Relabel` operations.

- There are $O(V^2)$ relabels total (each vertex's height is bounded by $O(V)$ and only increases — same style of argument as bounding relabels in general), so there are $O(V^2)$ phases.
- **Each phase contains at most $|V|$ calls to `DISCHARGE`.** Why: if a call to `DISCHARGE` does *not* relabel, the very next call moves strictly further down L (§4: no-op or one push, pointer only advances or stays), and L has length $< |V|$ — so this can happen at most $|V| - 1$ times before a relabel is forced, at which point the phase ends. Hence:

$$\text{total calls to } \texttt{DISCHARGE} = O(V^2) \text{ phases} \times O(V) \text{ calls/phase} = O(V^3).$$

That single inequality is the crux of the whole proof; everything else is bounding the *work per call*.

Now charge the internal work of `DISCHARGE` by type:

Source of work	Bound	Reasoning
Relabel operations	$O(VE)$	$O(V^2)$ relabels total, each costing $O(V)$ with a suitable implementation
Advancing `current` pointer	$O(VE)$	each vertex's pointer sweeps its neighbor list at most once per relabel of that vertex; summing $\deg(u) \times O(V)$ over all u gives $O(VE)$ by the handshaking lemma
Saturating pushes	$O(VE)$	between two saturating pushes on the same edge (u, v), u's height must rise by ≥ 2, independent of ordering

Source of work	Bound	Reasoning
Non-saturating pushes	$O(V^3)$	**at most one per call to** `DISCHARGE` (a non-saturating push drains $u.e$ to 0 and `DISCHARGE` returns immediately), and there are $O(V^3)$ calls
Outer while-loop bookkeeping (moves, list traversal)	$O(V^3)$	dominated by the $O(V^3)$ call count itself

Total: $O(V^3) + O(VE) = O(V^3)$, since $E = O(V^2) \Rightarrow VE = O(V^3)$.

Theorem 26.30. `RELABEL-TO-FRONT` runs in $O(V^3)$ time on any flow network.

One-line intuition for the whole argument: *relabels are cheap and rare ($O(V^2)$); everything else is only expensive because `DISCHARGE` might get called a lot — and the phase argument caps that call count at $O(V^3)$ using nothing but "each phase, without a relabel, can't outlast the list."*

---

10. Cheat sheet

State per vertex: height $h(u) \in \mathbb{N}$, excess $e(u) \geq 0$, neighbor-list pointer `current`.

Invariants: $h(s) = |V|$, $h(t) = 0$; residual edge $(u, v) \Rightarrow h(u) \leq h(v) + 1$; heights never decrease.

Push(u,v): legal iff u overflowing, $c_f(u, v) > 0$, $h(u) = h(v) + 1$. Moves $\min(e(u), c_f(u, v))$ downhill.

Relabel(u): legal iff u overflowing and no admissible edge leaves it. $h(u) \leftarrow 1 + \min\{h(v) : (u, v) \in E_f\}$.

Discharge(u): push/relabel u exclusively, repeatedly, until $e(u) = 0$. The main algorithm is just: while something is active, discharge it. Correct for *any* order of picking u; speed depends entirely on the order.

Relabel-to-front: discharge in list order; a vertex whose height just increased jumps to the front, and the next vertex processed is read *after* that jump (i.e. whoever the mover just displaced from the head). The list stays a topological sort of the admissible network $\Rightarrow$ correct, $O(V^3)$.

Why $O(V^3)$: $O(V^2)$ relabels $\Rightarrow$ $O(V^2)$ phases $\Rightarrow$ $O(V)$ discharges/phase $\Rightarrow$ $O(V^3)$ discharge calls $\Rightarrow$ everything charged per-call (like the one non-saturating push per call) costs $O(V^3)$

total; everything charged globally (relabels, saturating pushes, pointer advances) is only $O(VE) = O(V^3)$.