EDGE Algorithm Tutorial: ⟨Maximum Flow (Push–Relabel)⟩

Sanjana S. Mali
University of California, Davis
Davis, California, USA

Toluwanimi O. Odemuyiwa
University of California, Davis
Davis, California, USA

Serban D. Porumbescu
University of California, Davis
Davis, California, USA

Joel S. Emer
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA

John D. Owens
University of California, Davis
Davis, California, USA

1 Introduction

This document explains the push–relabel maximum-flow algorithm using the EDGE (Extended General Einsums) abstraction [3].

1.1 Why This Tutorial?

This tutorial belongs to the EDGE Zoo[1], a growing collection of worked examples that each express a well-known algorithm as an extended Einsum cascade. The Zoo targets a variety of algorithms. Each tutorial follows the same structure so that readers can compare the specifications with one another.

***Translating a known algorithm into EDGE.*.** We assume the reader already knows push–relabel. The emphasis here is on the *translation*. Section 3 runs one small network through preflow, push and relabel steps, and Section 4 runs the same network through the same steps expressed as tensors. Section 5 states the full cascade and Section 6 explains each line in both graph-algorithm terms and Einsum terms. Reading the two views against each other is meant to make the notation concrete in a way that just the specification cannot.

A textbook sequential algorithm, written as a bulk-synchronous one. Push–relabel is usually presented sequentially: *while some vertex is active, pick one, and either push or relabel it.* The cascade in Section 5 is instead bulk-synchronous — every active vertex acts in every generation — because that is what the Einsum machinery expresses naturally. Nothing about the algorithm requires the sequential reading; the `while` loop is a scheduling decision, and Section 8 recovers it as a cascade of its own.

Separating the algorithm from its implementation. Section 8 develops that point. Three cascades appear there — bulk-synchronous, predicated-sequential, and branching-sequential — and they form a short ladder in which each rung is exactly one decision. Reading them against one another separates what is algorithmic from what is a choice about when work happens, which is a large part of what the Zoo is for.

[1] See https://www.edge-lang.org/ for the Zoo and https://github.com/edge-einsums for the EDGE repositories.

2 Problem Statement

2.1 Problem Description

The maximum-flow problem asks how much can be shipped from a source vertex s to a sink vertex t through a capacitated network. Each directed edge (u, v) carries a capacity $c(u, v)$ bounding how much may travel along it, and a feasible flow must respect every capacity while conserving at every vertex other than s and t: what arrives must leave. The value of a flow is the net amount leaving the source, and the problem is to maximize it.

Push–relabel [2] solves this without ever searching for an augmenting path. Instead it maintains a *preflow* — a flow that is allowed to violate conservation by letting vertices hold a surplus — and pushes that surplus downhill until none is left. Two quantities drive it:

- the *excess* $e(u)$, how much flow has arrived at u and not yet left. A vertex with positive excess is *active*.
- the *distance* (or height) $d(u)$, a lower bound on how far u is from the sink in the residual network.

Flow may only move from u to v when the residual edge (u, v) still has capacity and $d(u) = d(v) + 1$ — strictly one level downhill. Such an edge is *admissible*. An active vertex with an admissible edge *pushes*; an active vertex with none is stuck and *relabels*, raising its own distance just high enough to open one up. The algorithm ends when no vertex other than s and t holds excess, at which point the preflow has become a flow, and that flow is maximum.

2.2 Inputs

The algorithm takes the following inputs:

- A directed graph $G = (V, E)$ where:
  - V is the set of vertices
  - E is the set of edges
  - each edge $(u, v) \in E$ has a capacity $c(u, v) \geq 0$
- A source vertex $s \in V$ and a sink vertex $t \in V$, with $s \neq t$

We make the following assumptions:

- Capacities are non-negative
- If an edge (u, v) is absent, it is treated as having capacity 0
- The network has no antiparallel pair (u, v) and (v, u) both present. This is the standard presentation and it keeps the residual initialization a disjoint case analysis; a network that has such a pair is first transformed by splitting one of the two edges.

2.3 Outputs

The output is a flow tensor F, where:

- $F[u][v]$ is the flow sent along edge (u, v)
- F is antisymmetric: $F[v][u] = -F[u][v]$, so one cell records both directions and a negative entry means flow runs the other way
- the value of the maximum flow is the net flow out of the source, $\sum_v F[s][v]$

2.4 Algorithmic Steps

The algorithm proceeds as follows.

1. Initialize. Set every distance to 0 except the source, which is lifted to $d(s) = |V|$ so that no flow is ever pushed back into it. Set all flow and all excess to 0.
2. Saturate the source. Send $c(s, v)$ along every edge leaving s. This is the *preflow*: the neighbours of the source are flooded with excess, and the reverse residual edges back toward the source open up.
3. Repeat until no internal vertex is active:
   - For an active vertex u with an admissible edge (u, v), *push* $\min(e(u), r(u, v))$ along it.
   - For an active vertex u with no admissible edge, *relabel* it to $1 + \min\{\, d(v) \mid r(u, v) > 0 \,\}$.
4. Return F.

For reference, Algorithm 1 gives a standard sequential presentation.

Note what the pseudocode does *not* say. It says "while some v has $e(v) > 0$", and never which such v. That freedom is real: the algorithm is correct for any choice, and different choices change only how much work is done. The cascade in Section 5 takes the freedom in the other direction and lets *every* active vertex act at once; Section 8 returns to the single-vertex reading and shows what it costs to express.

Algorithm 1 Push–Relabel

```
for each edge (u, v) ∈ E do
    f(u, v) ← 0;    r(u, v) ← c(u, v)
end for
for each vertex v ∈ V do
    d(v) ← 0;    e(v) ← 0
end for
d(s) ← |V|
for each neighbour v of s do                ▷ saturating preflow
    f(s, v) ← c(s, v);    f(v, s) ← −c(s, v)
    r(s, v) ← 0;    r(v, s) ← c(s, v)
    e(v) ← c(s, v);    e(s) ← e(s) − c(s, v)
end for
while some v ≠ s, t has e(v) > 0 do
    if some w has r(v, w) > 0 and d(v) = d(w) + 1 then
        Δ ← min(e(v), r(v, w))                ▷ push
        f(v, w) += Δ;    f(w, v) −= Δ
        r(v, w) −= Δ;    r(w, v) += Δ
        e(v) −= Δ;    e(w) += Δ
    else
        d(v) ← 1 + min{ d(w) | r(v, w) > 0 }    ▷ relabel
    end if
end while
return f
```

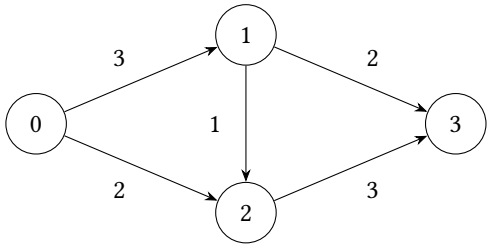

Figure 1. Example capacitated network. Source $s = 0$, sink $t = 3$.

3 Algorithm Execution Flow: Graph View

3.1 Example Network

Consider the network in Figure 1. It has four vertices and five capacitated edges.

- Vertices: $V = \{0, 1, 2, 3\}$
- Source vertex: $s = 0$
- Sink vertex: $t = 3$
- Internal vertices: 1 and 2

The edges and their capacities are:

- $0 \to 1$ with capacity 3
- $0 \to 2$ with capacity 2
- $1 \to 2$ with capacity 1
- $1 \to 3$ with capacity 2
- $2 \to 3$ with capacity 3

The two edges leaving the source carry total capacity $3 + 2 = 5$, and the two edges entering the sink carry total capacity $2 + 3 = 5$. Both cuts are saturated by a flow of 5, so the maximum flow is 5 and we know the answer before we start. What the trace below shows is *how* push–relabel discovers it.

3.2 Initialization

The algorithm begins by lifting the source and zeroing everything else.

- Every distance starts at zero, except the source, which is lifted to $|V| = 4$:

$$D_{i\equiv 1} = \{\, 0{:}4,\ 1{:}0,\ 2{:}0,\ 3{:}0 \,\}$$

- Flow and excess start empty:

$$F_{i\equiv 0} = \{\,\}, \qquad E_{i\equiv 0} = \{\,\}$$

Lifting the source to $|V|$ is what keeps flow from running back into it. No vertex can ever reach distance $|V|$ by relabelling while a path to the sink remains, so the admissibility rule $d(u) = d(v) + 1$ can never open an edge into s from a vertex that still has somewhere better to send its excess.

3.3 Preflow

The preflow step saturates both edges out of the source:

$$F_{1,0,1} = 3, \qquad F_{1,0,2} = 2$$

Because F is antisymmetric, this also records $F_{1,1,0} = -3$ and $F_{1,2,0} = -2$ on the reverse cells, which is what opens the residual edges back toward the source. The two internal vertices are now flooded:

$$E_1 = \{\, 1{:}3,\ 2{:}2 \,\}, \qquad Act_1 = \{\, 1,\ 2 \,\}$$

and the residual network is the capacity graph with the source row emptied and the reverse edges opened:

$$R_1 = \{\, (1,0){:}3,\ (1,2){:}1,\ (1,3){:}2,\ (2,0){:}2,\ (2,3){:}3 \,\}$$

Note that the source itself now holds $e(0) = -5$. Negative excess at the source is not a defect: it is the bookkeeping record of how much has been sent out, and it is exactly why the active set must exclude s and t. Were the termination test written on E rather than on the active set, it would never fire — at the end the sink holds $+5$ and the source -5, and E is not empty. Section 7 returns to this.

3.4 Push and Relabel

In the first generation both internal vertices sit at distance 0, and so do all of their neighbours. The rule $d(u) = d(v) + 1$ is satisfied nowhere, so neither vertex can push and both relabel to distance 1 — one above the lowest neighbour they can still reach.

That opens the edges to the sink. In the second generation vertex 1 sends 2 units along $1 \to 3$ and vertex 2 sends 2 units along $2 \to 3$. Vertex 1 had 3 units, so one is left over, and the edge it just used is now saturated: it relabels again, to distance 2. That lets it push its last unit across $1 \to 2$ in the third generation, handing vertex 2 one unit of excess, which vertex 2 forwards to the sink in the fourth. No internal vertex holds excess any more, and the cascade halts.

Summing the flow leaving the source gives

$$F_{0,1} + F_{0,2} = 3 + 2 = 5,$$

which matches the cut bound established above. The next section runs the same four generations again, as tensors.

4 Algorithm Execution Flow: EDGE Tensor View

4.1 Tensors for the Working Example

We reuse the network of Section 3, with $V = \{0, 1, 2, 3\}$, $s = 0$ and $t = 3$. In EDGE the network and the evolving state are tensors over two vertex ranks, U and V, both of shape $|V|$.

Capacity tensor C**.** $C_{u,v}$ holds the capacity of the directed edge $u \to v$, with the first index the row (tail) and the second the column (head). A missing edge is the empty value 0:

$C =$

$C_{u,v}$	v=0	v=1	v=2	v=3
u=0	·	3	2	·
u=1	·	·	1	2
u=2	·	·	·	3
u=3	·	·	·	·

Source and sink masks S, T**.** One-hot Boolean vectors over U: S_0 = True and T_3 = True. Their complements intersect to NST, the internal vertices $\{1, 2\}$ — the only ones permitted to be active.

Flow tensor F ***and residual tensor*** R**.** $F_{i,u,v}$ is the flow on $u \to v$ at generation i, antisymmetric so that $F_{i,v,u} = -F_{i,u,v}$. $R_{i,u,v}$ is the residual capacity. A saturated edge reaches 0, which *is* the empty value of R, so it simply disappears from the tensor rather than lingering as an explicit zero.

Excess E ***and distance*** D**.** $E_{i,u}$ is the excess held at u, empty value 0. $D_{i,u}$ is the distance, empty value $+\infty$ — not 0, because 0 is a legal and common distance and declaring it empty would make every distance-0 vertex vanish from the minimum in the relabel step.

4.2 Generation by Generation

Each generation lets every active vertex push along one admissible residual edge, then relabels any active vertex left with nothing admissible. Table 1 shows the four generations.

i	active	push ($u \to v$:amt)	relabel (u:D')	E_{i+1}
1	{1, 2}	none	1:1, 2:1	{1:3, 2:2}
2	{1, 2}	1→3:2, 2→3:2	1:2	{1:1}
3	{1}	1→2:1	none	{2:1}
4	{2}	2→3:1	none	{ } — stop

Table 1. Push–relabel on the network of Figure 1. The excess column lists internal vertices only; the source holds −5 and the sink accumulates +5 throughout.

Generation 1: both vertices are stuck. $Act_1 = \{1, 2\}$. Vertex 1 has residual edges to 0, 2 and 3 with distances 4, 0 and 0; admissibility needs $D_1 = D_v + 1$, so it would need a neighbour at distance 0 while standing at distance 1 — but it stands at 0. Nothing is admissible. The same holds for vertex

2. Both relabel:

$$MinNeiLbl_1 = \{\,1{:}0,\ 2{:}0\,\}, \qquad NewD_1 = \{\,1{:}1,\ 2{:}1\,\}$$

$$D_2 = \{\,0{:}4,\ 1{:}1,\ 2{:}1,\ 3{:}0\,\}$$

Note that vertex 1 takes the minimum over *all* its residual neighbours, which includes vertex 0 at distance 4. The minimum is 0, from the sink, so the relabel is to 1 and not to 5 — the source's height keeps it out of contention without having to exclude it explicitly.

Generation 2: both push, one gets stuck again. With $D_1 = D_2 = 1$ and $D_3 = 0$, both edges into the sink are admissible. Vertex 1 pushes $\min(3, 2) = 2$ on $1 \to 3$ and vertex 2 pushes $\min(2, 3) = 2$ on $2 \to 3$:

$$\delta_2 = \{\,(1,3){:}2,\ (2,3){:}2\,\}, \qquad E_3 = \{\,1{:}1\,\}$$

Vertex 2 is emptied. Vertex 1 keeps 1 unit, and its edge to the sink is now saturated and gone from R. Its remaining residual edges go to 0 (distance 4) and 2 (distance 1), neither admissible from distance 1, so it relabels to $\min(4, 1) + 1 = 2$.

This is the generation that shows why the cascade recomputes the admissible mask *after* the push. At the start of the generation vertex 1 had an admissible edge; by the end it does not, because it used it up. Only a post-push test can see that.

Generations 3 and 4: the last unit walks to the sink. From distance 2, vertex 1 finds $1 \to 2$ admissible ($D_1 = 2 = D_2 + 1$) and pushes its last unit, which saturates that edge and hands vertex 2 one unit of excess. In generation 4 vertex 2, still at distance 1 with the sink at 0, pushes it the rest of the way.

4.3 Termination

After generation 4 no internal vertex holds excess:

$$Act_5 = \{\,\} \qquad \Longrightarrow \qquad \|Act_5\| \equiv 0$$

and the cascade halts. The final flow is

$$F = \{\,(0,1){:}3,\ (0,2){:}2,\ (1,2){:}1,\ (1,3){:}2,\ (2,3){:}3\,\}$$

(plus the antisymmetric mirror of each entry), with value $F_{0,1} + F_{0,2} = 5$. Every edge out of the source and every edge into the sink is saturated, which certifies optimality directly.

The next sections generalize this run and give the full EDGE formulation.

5 EDGE Expression

Both vertex ranks U and V have shape $|V|$; the shape is spelled out once on C and left implicit afterwards. The full EDGE cascade for bulk-synchronous push–relabel is as follows.

▹ **Tensors**

$$\begin{aligned}
C^{U\equiv|V|,\,V\equiv|V|} &\to \text{Integer},\ \text{empty} = 0\\
F^{I,\,U,\,V} &\to \text{Integer},\ \text{empty} = 0\\
R^{I,\,U,\,V} &\to \text{Integer},\ \text{empty} = 0\\
E^{I,\,U} &\to \text{Integer},\ \text{empty} = 0\\
D^{I,\,U} &\to \text{Integer},\ \text{empty} = +\infty\\
S^{U},\ T^{U} &\to \text{Boolean},\ \text{empty} = \text{False}\\
NST^{U} &\to \text{Boolean},\ \text{empty} = \text{False}\\
Act^{I,\,U} &\to \text{Boolean},\ \text{empty} = \text{False}
\end{aligned}$$

▹ **Initializations**

$$\begin{aligned}
C_{u,v} &\to \langle\text{user-specified}\rangle\\
F_{i\equiv 0,u,v} = 0, \quad E_{i\equiv 0,u} &= 0\\
D_{i\equiv 1,u} = 0, \quad D_{i\equiv 1,u:u=s} &= |V|\\
S_{u:u=s}, \quad T_{u:u=t} &= \text{True}
\end{aligned}$$

▹ **Preflow**

$$FS_{i\equiv 1,u,v} = S_u \cdot C_{u,v} :: \bigwedge * (\cap) \tag{5.1}$$

[Saturate every edge leaving the source]

$$F_{i\equiv 1,u,v} = FS_{i\equiv 1,u,v} \cdot FS_{i\equiv 1,v,u} :: \bigwedge -(\cup) \tag{5.2}$$

[Mirror it as antisymmetric flow]

$$E_{i\equiv 1,v} = F_{i\equiv 1,u,v} :: \bigvee + (\cup) \tag{5.3}$$

[Excess is net inflow]

$$R_{i\equiv 1,u,v} = \begin{cases} 0 & u = s \\ C_{v,u} & v = s \ \wedge\ u \neq s \\ C_{u,v} & \text{otherwise} \end{cases} \tag{5.4}$$

[Source arcs saturated, reverse arcs opened]

$$NST_u = \neg S_u \cdot \neg T_u :: \bigwedge \text{AND} (\cap) \tag{5.5}$$

[The internal vertices]

$$Act_{i\equiv 1,u} = NST_u \cdot^{1} (E_{i\equiv 1,u} \cdot^{2} 0)_{i\equiv 1,u} :: \bigwedge^{1} \leftarrow (\cap) \quad \bigwedge^{2} > (\cap) \tag{5.6}$$

[Seed the active set]

▹ **Extended Einsums**

$$ActR_{i,u,v} = Act_{i,u} \cdot R_{i,u,v} :: \bigwedge \leftarrow (\cap) \tag{5.7}$$

[Residual edges leaving an active vertex]

$$Lbl_{i,u,v} = D_{i,u} \cdot (D_{i,v} + 1)_{i,v} :: \bigwedge \equiv (\cap) \tag{5.8}$$

[Edges running exactly one level downhill]

$$Adm_{i,u,v} = ActR_{i,u,v} \cdot Lbl_{i,u,v}$$

$$:: \bigwedge \mathrm{AND}(\cap) \tag{5.9}$$

[Admissible edges]

$$PushCand_{i,u,v^*} = Adm_{i,u,v} \lll_{v^*} \mathbb{1}(\text{pick-admissible-edge}) \tag{5.10}$$

[One admissible edge per pushing vertex]

$$\delta_{i,u,v} = (E_{i,u} \cdot^1 R_{i,u,v})_{i,u,v} \cdot^2 PushCand_{i,u,v} :: \bigwedge^1 \min(\cap) \ \bigwedge^2 \leftarrow (\cap) \tag{5.11}$$

[Push the smaller of excess and residual]

$$F_{i+1,u,v} = (F_{i,u,v} \cdot^1 \delta_{i,u,v})_{i,u,v} \cdot^2 \delta_{i,v,u} :: \bigwedge^1 +(\cup) \ \bigwedge^2 -(\cup) \tag{5.12}$$

[Add forward, subtract on the reverse edge]

$$InPush_{i,v} = \delta_{i,u,v} :: \bigvee +(\cup) \tag{5.13}$$

[Flow arriving at each vertex]

$$OutPush_{i,u} = \delta_{i,u,v} :: \bigvee +(\cup) \tag{5.14}$$

[Flow leaving each vertex]

$$E_{i+1,u} = (E_{i,u} \cdot^1 InPush_{i,u})_{i,u} \cdot^2 OutPush_{i,u} :: \bigwedge^1 +(\cup) \ \bigwedge^2 -(\cup) \tag{5.15}$$

[Gains what arrived, loses what left]

$$R_{i+1,u,v} = (R_{i,u,v} \cdot^1 \delta_{i,u,v})_{i,u,v} \cdot^2 \delta_{i,v,u} :: \bigwedge^1 -(\cup) \ \bigwedge^2 +(\cup) \tag{5.16}$$

[Forward residual shrinks, reverse grows]

$$Adm_{i+1,u,v} = R_{i+1,u,v} \cdot Lbl_{i,u,v} :: \bigwedge \to (\cap) \tag{5.17}$$

[Still admissible after the push]

$$HasAdm_{i+1,u} = Adm_{i+1,u,v} :: \bigvee \mathrm{OR}(\cup) \tag{5.18}$$

[Anywhere left to push?]

$$Act_{i+1,u} = NST_u \cdot^1 (E_{i+1,u} \cdot^2 0)_{i+1,u} :: \bigwedge^1 \leftarrow (\cap) \ \bigwedge^2 > (\cap) \tag{5.19}$$

[Recompute the active set]

$$Rel_{i+1,u} = Act_{i+1,u} \cdot \neg HasAdm_{i+1,u} :: \bigwedge \mathrm{AND}(\cap) \tag{5.20}$$

[Still active but stuck: these relabel]

$$NeiLbl_{i,u,v} = (R_{i+1,u,v} \cdot^1 Rel_{i+1,u})_{i,u,v} \cdot^2 D_{i,v} :: \bigwedge^1 \leftarrow (\cap) \ \bigwedge^2 \to (\cap) \tag{5.21}$$

[Distances of the residual neighbours]

$$MinNeiLbl_{i,u} = NeiLbl_{i,u,v} :: \bigvee \min(\cup) \tag{5.22}$$

[The lowest such distance]

$$NewD_{i,u} = (MinNeiLbl_{i,u} \cdot 1)_{i,u} :: \bigwedge +(\cap) \tag{5.23}$$

[One above it]

$$D_{i+1,u} = D_{i,u} \cdot NewD_{i,u} :: \bigwedge \ll(\cup) \tag{5.24}$$

[Relabelled take NewD, the rest keep D]

$$\diamond : \ \|Act_{i+1}\| \equiv 0 \tag{5.25}$$

[Stop when no internal vertex holds excess]

Cascade 1. Push–Relabel Maximum Flow EDGE Cascade (bulk-synchronous)

6 EDGE Expression Walk-Through

We now walk through the cascade of Section 5 step by step. The goal is to connect each tensor operation to the "push excess downhill, lift the stuck" view of the algorithm, while also saying what each line means as an Einsum computation.

6.1 Preflow

Saturate the source.

$$FS_{i\equiv 1,u,v} = S_u \cdot C_{u,v} :: \bigwedge *(\cap)$$

Graph meaning: send the full capacity along every edge leaving s.

Einsum meaning: for a point (u, v) of FS, take S at u and C at (u, v). The intersection keeps only points where both exist, and since S is one-hot, only the $u = s$ row of C survives. The Boolean multiplies as 1, so the surviving values are the capacities themselves.

Mirror it as antisymmetric flow.

$$F_{i\equiv 1,u,v} = FS_{i\equiv 1,u,v} \cdot FS_{i\equiv 1,v,u} :: \bigwedge -(\cup)$$

Graph meaning: record $f(s, v) = c(s, v)$ forward and $f(v, s) = -c(s, v)$ on the reverse edge.

Einsum meaning: the same tensor is read at a point and at its transpose, and the two are subtracted. At (s, v) this gives $c(s, v) - 0$; at (v, s) it gives $0 - c(s, v)$. One expression produces both directions, which is what makes F antisymmetric by construction rather than by convention.

Excess is net inflow.

$$E_{i\equiv 1,v} = F_{i\equiv 1,u,v} \\ :: \bigvee +(\cup)$$

Graph meaning: each vertex holds what arrived minus what left.

Einsum meaning: reduce over rank u, summing F at all points (u, v) for each v. Outflow is already stored as negative entries, so a single sum gives net inflow — no separate subtraction is needed. At the source this correctly yields $-\sum_v c(s, v)$.

Build the residual network.

$$R_{i\equiv 1,u,v} = \begin{cases} 0 & u = s \\ C_{v,u} & v = s \ \wedge \ u \neq s \\ C_{u,v} & \text{otherwise} \end{cases}$$

Graph meaning: the source's own edges are saturated and so carry no residual capacity, while the reverse edges back toward the source open up to the amount just sent. Every other edge keeps its capacity.

Einsum meaning: a three-way case on the iteration space. The cases are disjoint — the $v = s$ arm carries an explicit $u \neq s$ guard — so the result does not depend on evaluation order, including at a source self-loop.

6.2 Choosing Who Acts

The internal vertices.

$$NST_u = \neg S_u \cdot \neg T_u \\ :: \bigwedge \text{AND}(\cap)$$

Neither source nor sink. Only these may be active: the source holds negative excess by construction and the sink accumulates the answer, and neither should ever push.

The active set.

$$Act_{i,u} = NST_u \cdot^1 (E_{i,u} \cdot^2 0)_{i,u} \\ :: \bigwedge{}^1 \leftarrow (\cap) \quad \bigwedge{}^2 > (\cap)$$

Graph meaning: a vertex is active when it is internal and holds positive excess.

Einsum meaning: the inner sub-merge tests $E_{i,u} > 0$, and the outer take-left keeps NST's Boolean at the coordinates that survive both. Note the nesting: the two operation labels $\cdot^1$ and $\cdot^2$ pair with $\bigwedge^1$ and $\bigwedge^2$, so the comparison happens first and its result is then intersected with NST.

This line appears twice in the cascade — once pinned to $i \equiv 1$ in the preflow and once at $i + 1$ in the loop body. The first only *seeds* the iteration; every round after the first reads the copy the relabel step produces. Section 7 explains why a bulk-synchronous cascade needs both.

6.3 Admissibility

$$ActR_{i,u,v} = Act_{i,u} \cdot R_{i,u,v} \\ :: \bigwedge \leftarrow (\cap)$$

keeps the residual edges that leave an active vertex. No explicit > 0 test is needed: a saturated edge has value 0, which is R's empty value, so it has already dropped out and the intersection skips it. This is the first of several places where a predicate is carried by an empty value rather than by a comparison.

$$Lbl_{i,u,v} = D_{i,u} \cdot (D_{i,v} + 1)_{i,v} \\ :: \bigwedge \equiv (\cap)$$

is True exactly where $D(u) = D(v) + 1$. Writing it in this direction — $D_v + 1$ rather than $D_u - 1$ — matters: distances are non-negative and D's empty value is $+\infty$, so adding can never collide with the empty value, while subtracting from a vertex at distance 0 could.

Their conjunction

$$Adm_{i,u,v} = ActR_{i,u,v} \cdot Lbl_{i,u,v} \\ :: \bigwedge \text{AND}(\cap)$$

is the set of admissible edges: residual capacity remaining, running one level downhill, out of a vertex that has something to send.

6.4 Push

$$PushCand_{i,u,v^*} = Adm_{i,u,v} \\ \lll_{v^*} \mathbb{1}(\text{pick-admissible-edge})$$

The populate action keeps one admissible neighbour v for each pushing vertex u and drops the rest of that u-row. Any admissible neighbour is a valid choice, which is the populate's point: the algorithm is correct whichever is taken.

One edge per vertex is *required*, not merely convenient. Every admissible edge of u would otherwise compute its own min against the same $E_{i,u}$, and u would send out more than it holds.

$$\delta_{i,u,v} = (E_{i,u} \cdot^1 R_{i,u,v})_{i,u,v} \cdot^2 PushCand_{i,u,v} \\ :: \bigwedge{}^1 \min(\cap) \quad \bigwedge{}^2 \leftarrow (\cap)$$

The amount pushed is the smaller of the vertex's excess and the edge's residual capacity, restricted to the chosen edge. Taking min against the residual is what keeps the flow feasible; taking it against the excess is what keeps the preflow from going negative.

Flow is then adjusted on the chosen edges and their reverses:

$$F_{i+1,u,v} = (F_{i,u,v} \cdot^1 \delta_{i,u,v})_{i,u,v} \cdot^2 \delta_{i,v,u} \\ :: \bigwedge{}^1 +(\cup) \quad \bigwedge{}^2 -(\cup)$$

The distance rule makes $u \rightarrow v$ and $v \rightarrow u$ mutually exclusive within one generation — both would require

$D(u) = D(v) + 1$ and $D(v) = D(u) + 1$ — so at most one of $\delta_{i,u,v}$ and $\delta_{i,v,u}$ is ever present. The + and the − therefore land on a cell and its mirror in a single write, with no possibility of double counting.

Excess is updated by what each vertex received against what it sent,

$$\begin{aligned} InPush_{i,v} &= \delta_{i,u,v} \\ &:: \bigvee + (\cup) \\ OutPush_{i,u} &= \delta_{i,u,v} \\ &:: \bigvee + (\cup) \end{aligned}$$

$$\begin{aligned} E_{i+1,u} &= (E_{i,u} \cdot^1 InPush_{i,u})_{i,u} \cdot^2 OutPush_{i,u} \\ &:: \bigwedge^1 + (\cup) \quad \bigwedge^2 - (\cup) \end{aligned}$$

The two reductions differ only in which rank they reduce — *InPush* is indexed by v so u is summed away, and *OutPush* the reverse. The + in *InPush* is load-bearing in this cascade: several vertices may push into the same v in one generation, and their contributions must accumulate.

The residual updates symmetrically, used capacity subtracted forward and reverse capacity added back:

$$\begin{aligned} R_{i+1,u,v} &= (R_{i,u,v} \cdot^1 \delta_{i,u,v})_{i,u,v} \cdot^2 \delta_{i,v,u} \\ &:: \bigwedge^1 - (\cup) \quad \bigwedge^2 + (\cup) \end{aligned}$$

6.5 Relabel

The relabel arm runs on the *post-push* state. Using the updated residual, recompute the admissible mask

$$\begin{aligned} Adm_{i+1,u,v} &= R_{i+1,u,v} \cdot Lbl_{i,u,v} \\ &:: \bigwedge \to (\cap) \end{aligned}$$

— distances have not moved yet, so *Lbl* is still at generation i — and ask whether each vertex still has anywhere to go:

$$\begin{aligned} HasAdm_{i+1,u} &= Adm_{i+1,u,v} \\ &:: \bigvee \mathrm{OR}(\cup) \end{aligned}$$

The active set is likewise recomputed on the updated excess,

$$\begin{aligned} Act_{i+1,u} &= NST_u \cdot^1 (E_{i+1,u} \cdot^2 0)_{i+1,u} \\ &:: \bigwedge^1 \leftarrow (\cap) \quad \bigwedge^2 > (\cap) \end{aligned}$$

and a vertex relabels when it is still active but stuck:

$$\begin{aligned} Rel_{i+1,u} &= Act_{i+1,u} \cdot \neg HasAdm_{i+1,u} \\ &:: \bigwedge \mathrm{AND}(\cap) \end{aligned}$$

This is the `else` branch of the pseudocode, expressed as a mask rather than as control flow. Nothing selects between two code paths; both arms are written unconditionally, and the one that should not fire is empty.

For the relabelling vertices, gather the distances of their residual neighbours, take the minimum, add one, and write it back:

$$\begin{aligned} NeiLbl_{i,u,v} &= (R_{i+1,u,v} \cdot^1 Rel_{i+1,u})_{i,u,v} \cdot^2 D_{i,v} \\ &:: \bigwedge^1 \leftarrow (\cap) \quad \bigwedge^2 \to (\cap) \end{aligned}$$

$$\begin{aligned} MinNeiLbl_{i,u} &= NeiLbl_{i,u,v} \\ &:: \bigvee \min(\cup) \\ NewD_{i,u} &= (MinNeiLbl_{i,u} \cdot 1)_{i,u} \\ &:: \bigwedge + (\cap) \end{aligned}$$

$$\begin{aligned} D_{i+1,u} &= D_{i,u} \cdot NewD_{i,u} \\ &:: \bigwedge \ll (\cup) \end{aligned}$$

The update operator $\ll$ takes the right operand where it is present and falls back to the left, so relabelled vertices take *NewD* and every other vertex keeps the distance it had. For a vertex that is not relabelling, *NeiLbl* is empty, so *MinNeiLbl* is $+\infty$ — its empty value — and *NewD* is empty too. The whole arm is a no-op for it without any additional gate.

6.6 Termination

$$\diamond : \ \|Act_{i+1}\| \equiv 0$$

Graph meaning: no internal vertex holds excess. Every unit of preflow has either reached the sink or been returned to the source, the preflow has become a flow, and that flow is maximum.

Einsum meaning: count the non-empty points of *Act* and stop when there are none. The test is on the active set and not on E, which matters: at termination the sink holds the whole flow and the source holds minus it, so $\|E\| \equiv 0$ would never fire.

7 Other Notes

7.1 Empty Values Are Not All Zero

A stored value equal to a tensor's declared empty value is *absent*. That single rule does a surprising amount of work in this cascade, and it only works because the empty values are chosen per tensor rather than globally.

- C, F, R, E, δ: empty = 0. A saturated residual edge *is* zero, so it disappears from R and every intersection involving R skips it — which is why admissibility needs no explicit $r(u, v) > 0$ guard. Likewise a vertex with no excess disappears from E.
- D, *NeiLbl*, *MinNeiLbl*, *NewD*: empty = $+\infty$. These hold distances, and distance 0 is both legal and common — the sink sits there for the entire run. Declaring 0 empty would drop every distance-0 neighbour before the minimum in the relabel step ever saw it, and the algorithm would relabel to the wrong heights.
- S, T, *NST*, *Act*, *Adm*, *Rel*: empty = False, the ordinary reading for a predicate.

The $+\infty$ on *MinNeiLbl* does double duty. It is the right empty value for a distance, and it is also what makes the relabel arm vanish for a vertex that pushed: *NeiLbl* is empty for that vertex, so the reduction leaves $+\infty$, so *NewD* is empty, so $\ll$ keeps the old distance. No second gate is needed.

7.2 Why the Cascade Recomputes After the Push

Three Einsums in the loop body are recomputations of things the generation already had: *Adm* at $i+1$, *HasAdm* at $i+1$, and *Act* at $i+1$. They are there because bulk-synchronous evaluation has no read-after-write within a generation. The relabel arm has to ask "is this vertex *still* stuck, now that the pushes have happened?", and the only way to ask it is to recompute the mask on R_{i+1}.

That is also what makes a generation of this cascade not quite an iteration of the `while` loop. A vertex that makes a saturating push, still holds excess, and now has nothing admissible will push *and* relabel in the same generation. Both are legal push–relabel steps and the result is correct, but the round structure differs from the textbook one. Section 8 takes this apart.

7.3 Why One Edge per Vertex

PushCand selects a single admissible edge per pushing vertex. It is worth being explicit that this is a correctness requirement and not a performance heuristic.

Suppose u has excess 3 and two admissible edges, each with residual capacity 3. Without the populate, δ would take $\min(3,3)=3$ on *both*, and $OutPush_u$ would sum them to 6 — twice what u holds. The excess would go negative and the flow would become infeasible.

Apportioning the excess across several edges instead of choosing one is possible in principle, but it is a different algorithm: it needs a division rule, and the division rule would have to appear in the specification. Selecting one edge keeps the specification silent about something the algorithm is silent about.

7.4 Negative Excess at the Source

After the preflow the source holds $e(s) = -\sum_v c(s,v)$, and it keeps a negative excess for the whole run. This is not an artifact to be cleaned up: it is the record of how much has been shipped, and it is what makes the final flow value readable as $\sum_v F_{s,v}$.

It does, however, mean the termination test cannot be written on E. At the end the sink holds $+5$ and the source -5 in our example, so E is very much non-empty. The test is on *Act*, which excludes both by intersecting with *NST*, and that is the only reason it ever fires.

7.5 What the Populate Marks

Every populate in an EDGE cascade marks a place where the algorithm said "pick one and do it". In push–relabel there is exactly one such place in the bulk-synchronous form, *PushCand*, and Section 8 adds a second, *Sel*, to recover the sequential reading. Read the other way round, the bulk-synchronous cascade is the sequential one with a selector *removed*.

8 Algorithmic Variants

The cascade of Section 5 is bulk-synchronous: every active vertex acts in every generation. The textbook algorithm is sequential: one vertex acts at a time. Both are push–relabel, and both are correct.

This section writes down the sequential readings as cascades of their own. They form a short ladder, and the point of the ladder is that each rung is exactly *one* decision.

$$\text{parallel} \xrightarrow{\text{add a selector}} \text{predicated} \xrightarrow{\text{decide before the push}} \text{branching}$$

8.1 Rung 1: Add a Selector

The parallel cascade lets *Act* — a *set* — drive everything downstream. To act on one vertex instead, reduce that set to a single coordinate with a populate:

$$Sel_{i,u^*} = Act_{i,u} \lll_{u^*} \mathbb{1}(\text{select-any-vertex}) \tag{8.1}$$

[Added: reduce the active set to one vertex]

$$ActR_{i,u,v} = Sel_{i,u} \cdot R_{i,u,v} :: \bigwedge \leftarrow (\cap) \tag{8.2}$$

[Changed: reads Sel, not Act]

$$Rel_{i+1,u} = Act_{i+1,u} \cdot^1 \neg HasAdm_{i+1,u} \cdot^2 Sel_{i,u} :: \bigwedge^1 \text{AND}(\cap) \bigwedge^2 \text{AND}(\cap) \tag{8.3}$$

[Changed: gated by Sel]

Cascade 2. The selector, and the two Einsums that read it

That is the entire first rung: **one Einsum added, two changed**. Every other line of the cascade is unchanged, character for character. The name `select-any-vertex` is doing real work — the pseudocode says only "while some v has $e(v) > 0$" and never which, so any selection rule is faithful, and the populate says exactly that and no more.

This cascade is correct, but it is *predicated* rather than branching. Both arms still evaluate in every generation, and emptiness masks one of them out. The consequence is subtle: a generation is no longer an iteration of the `while` loop. A vertex that makes a saturating push, still holds excess, and now has nothing admissible will be caught by the post-push re-test and relabel in the *same* generation — two operations in one round.

8.2 Rung 2: Decide Before the Push

To make a generation one operation, ask the admissibility question *before* anything is written, and let it decide which arm runs:

$$HasAdm_{i,u} = Adm_{i,u,v} :: \bigvee \mathrm{OR}(\cup) \tag{8.4}$$

[Retimed to i: asked before the push]

$$Rel_{i,u} = Sel_{i,u} \cdot^{1} \neg HasAdm_{i,u} :: \bigwedge^{1} \mathrm{AND}(\cap) \tag{8.5}$$

[Retimed: the branch, before any write]

$$NeiLbl_{i,u,v} = (R_{i,u,v} \cdot^{1} Rel_{i,u})_{i,u,v} \cdot^{2} D_{i,v} :: \bigwedge^{1} \leftarrow (\cap) \quad \bigwedge^{2} \rightarrow (\cap) \tag{8.6}$$

[Retimed: reads R_i, the state decided from]

$$\diamond : \ \|Act_i\| \equiv 0 \tag{8.7}$$

[Retimed: tests the round's own Act]

Cascade 3. Moving the decision ahead of the writes

and **drop** the two post-push recomputations, which are now dead:

$Adm_{i+1,u,v}$ (recomputed from R_{i+1}),
$Act_{i+1,u}$ (recomputed after the push).

That is the second rung: **two dropped and three retimed**. Nothing is added. The arms are now mutually exclusive by construction — *Rel* is non-empty exactly when *Adm* is empty, and δ is derived from *Adm* — so each generation performs exactly one operation, and a generation *is* an iteration of the `while` loop.

Reading the relabel arm from R_i rather than R_{i+1} is not an optimization. The vertex did not push, so the two are equal; writing R_i states that the decision and the action use one state.

8.3 What the Ladder Costs

cascade	Einsums	rounds	operations
parallel (bulk-sync.)	24	4	7
predicated sequential	25	5	5
branching sequential	23	5	5

Table 2. The three cascades on the network of Figure 1. All three compute a maximum flow of 5.

Two things in Table 2 are worth reading carefully.

First, **the rung that changes the algorithm makes the cascade shorter, and the rung that only adds a selector makes it longer.** Adding *Sel* costs an Einsum; moving the decision earlier saves two.

Second, **this network does not separate the two sequential forms**. Both run in five rounds and perform five operations, because the distinguishing case — a saturating push that leaves its vertex stuck in the same round — never arises here. On the six-vertex network of CLRS Figure 26.1 [1] it does, three times: the predicated cascade packs 29 operations into 26 rounds, while the branching cascade takes 29 rounds for the same 29 operations in the same order. A small example is enough to state the transformation and not always enough to exhibit its consequences.

8.4 Where the Difference Lives

Against the parallel cascade, the branching cascade differs in eight places and agrees in nineteen. Every difference falls into one of the two rungs:

- **the selector** — *Sel* added; *ActR* and *Rel* read it;
- **the retiming** — *HasAdm*, *NeiLbl* and the stopping condition move to generation i; the two post-push recomputations disappear.

Everything else — the whole preflow, the whole push arm, the whole write-back to F, R, E and D — is the same expression in all three cascades. What sequentialising changes is how the cascade *decides*, not how it *writes*.

8.5 A Note on Rank Contraction

Once *Sel* is one-hot, every tensor masked by it has a U rank of occupancy at most one, and that rank can be contracted away: the chosen vertex's excess, height and admissibility test become 0-D, and its arcs become 1-D over V.

Doing so is a presentation choice and not a further rung. It changes no operation, no round and no value, but it roughly doubles the number of expressions that differ textually from the parallel cascade — replacing *ActR*, *Lbl*, *Adm*, *PushCand* and δ with 1-D counterparts and introducing three Einsums that exist only to project and re-expand. We state the sequential cascades with their ranks intact so that the eight differences above are the only ones a reader has to account for.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[2] Andrew V. Goldberg and Robert E. Tarjan. 1988. A New Approach to the Maximum-Flow Problem. *J. ACM* 35, 4 (oct 1988), 921–940. doi:10.1145/48014.61051

[3] Toluwanimi O. Odemuyiwa, Serban D. Porumbescu, Joel S. Emer, and John D. Owens. 2026. The EDGE Language: Extended General Einsums for Graph Algorithms. arXiv:2404.11591v2 [cs.DS] https://arxiv.org/abs/2404.11591v2