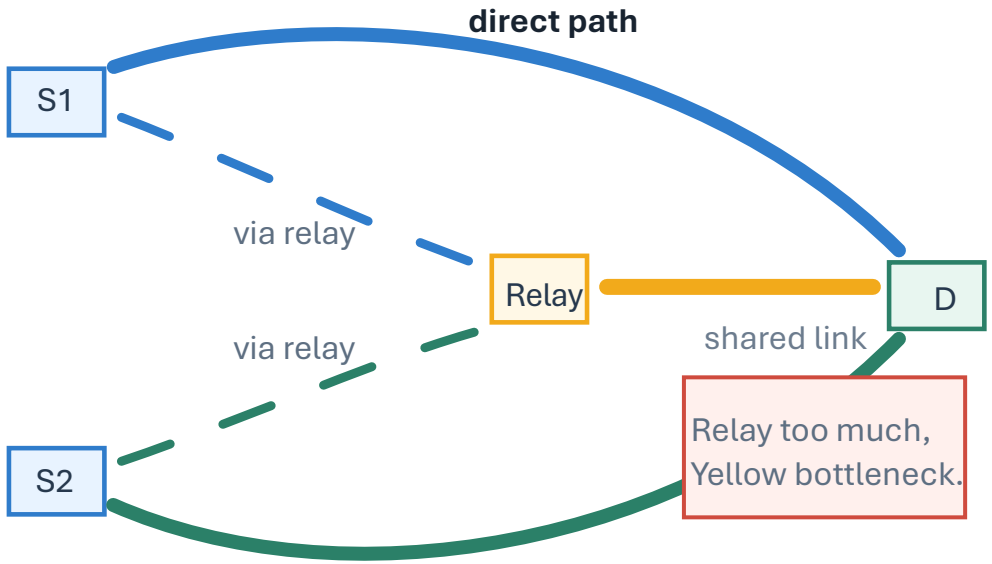
nsdi '26

NSDI '26 PAPER PRESENTATION

A Fast Solver-Free Algorithm for Traffic Engineering in Large-Scale Data Center Networks



Sequential Source-Destination Optimization (SSDO)

solver-free | coupling-aware | anytime

Yingming Mao[1 2], Qiaozhu Zhai[1] , Ximeng Liu[3], Zhen Yao[4], Xia Zhu [4], Yuzhou Zhou[1]

[1] Xi'an Jiaotong University [2] Shanghai Innovation Institute [3] Shanghai Jiao Tong University [4] Huawei

西安交通大學 XI'AN JIAOTONG UNIVERSITY

上海创智学院 Shanghai Innovation Institute

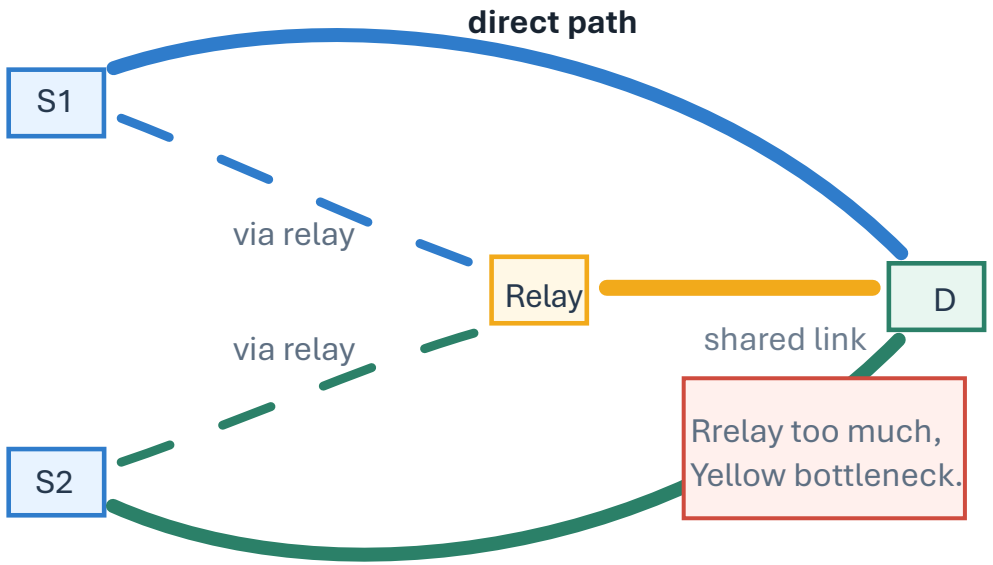
上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

HUAWEI

Why Traffic Engineering Is Necessary?

TE continuously maps changing demand onto shared network capacity.

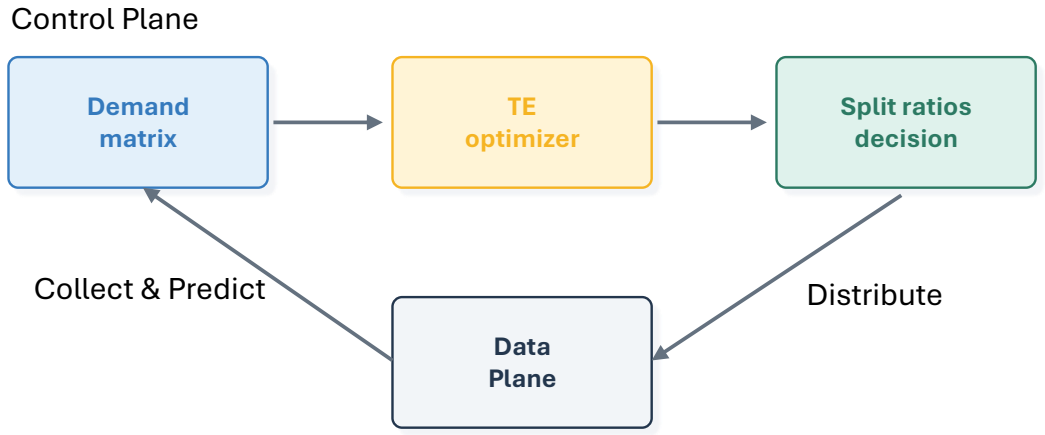
What does Traffic Engineering decide?



How should S1 and S2 split traffic?

TE decides the split ratio of each SD, and these split ratios jointly affect system performance, requiring coordinated optimization.

How does Traffic Engineering work?



- The above cycle repeats every **3 to 5 minutes**, and the TE optimizer must provide a feasible solution within the time limit.
- If the solution quality is too poor, **congestion** may occur.
- If the solving speed is too slow and **no feasible solution** is generated, the system will fall back to basic ECMP, which will also lead to **congestion**.

A good TE solver should make full use of these 3-5 minutes to obtain a solution with the highest possible quality!

TE Is a Linear Program, But Scale Makes It Painful

The hard part is not the formula; it is the number of coupled split-ratio variables.

TE Formulation

$$\min_{f_{ikj} \in \mathcal{R}} u$$

$$\text{s.t.} \begin{cases} f_{ikj} \geq 0, \quad f_{iki} = 0, \quad f_{iij} = 0, & \forall i, j, k \in V, \\ f_{ikj} = 0, & \forall (i, k, j) \notin \mathcal{P}, \\ \sum_{k \in V} f_{ikj} = 1, & \forall i \neq j \in V, \\ \dfrac{\sum_{k \in V} f_{ijk} \cdot D_{ik} + \sum_{k \in V} f_{kij} \cdot D_{kj}}{c_{ij}} \leq u, & \forall i \neq j \in V. \end{cases}$$

- f_{ikj} denotes the split ratio of path $i -> k -> j$ for demand D_{ij}
- f_{ijj} denotes the split ratio of direct path $i -> j$ for demand D_{ij}
- c_{ij} denotes the capacity of link $i -> j$
- u denotes minimize maximum link utilization (MLU), which is a widely adopted metric in TE.

Standard LP | **Theoretical optimal** | **Commercial solver**

TE is a Trivial Problem?

TE Scalability

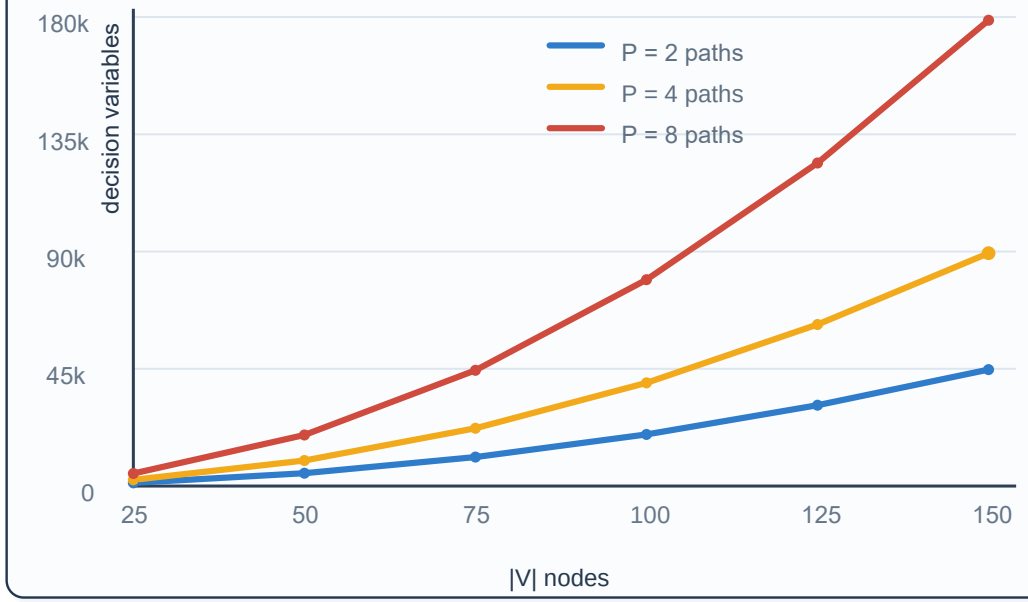
Variables

$$|V| \times (|V|\text{-}1) \times \text{paths}$$

Examples

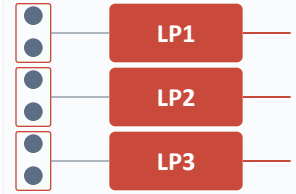
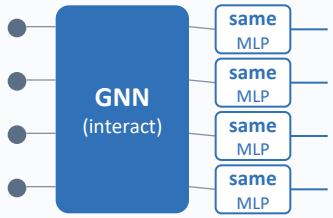
150 nodes × **149** destinations × **4** paths = **89,400**

More nodes and more candidate paths make the LP large even before solving it.



Existing TE: Terrible Trade-off

They simplify the coupled TE decision in different ways, but none preserves quality and scalability together.

	Direct (single solver)	Parallel (split into K)
LP exact solve	 **Commercial solver** Speed: Slow · Quality: Optimal · Scale: Fails • Theoretically optimal • No optima within time limit • Often not even feasible at scale	 K small subproblems, no coupling **Split into K subproblems** Speed: Mixed · Quality: Mixed · Scale: Mixed • Small K: still slow • Large K: quality collapses • Hard to choose K
DL learn mapping	 output dim = #paths (explodes at scale) **Learn demand→routing map** Speed: Fast · Quality: Approx · Scale: OOM • Fast inference • Bounded error vs optimal • Crashes when scale grows	 graph mixes; per-flow MLP shares weights **Shared MLP per flow** Speed: Fast · Quality: Lossy · Scale: Capped • Supports larger scale • Larger error from sharing • GNN still has a ceiling

Direct hits the scale wall → forced to go Parallel → hits the quality wall.

We need a method that scales without paying the parallel-coupling tax.

Sequential Optimization Looks Wrong

The first reaction is right — if every step is an LP, serial time is additive. Two design choices fix that.

Parallel TE

All SDs solved at once → 1 timestep

Looks fast ✓

Workers ignore each other → conflicts on shared links

Sequential SSDO

One SD at a time → N timesteps

Looks slow ✗

Each step sees current load → no conflicts, lower MLU

Our insight

Make each subproblem so cheap that N sequential steps still beat 1 parallel step — in time and in quality.

Choose a Subproblem That's Easy to Search

Fix every demand except one. The remaining problem becomes a monotone feasibility search over target MLU u.

Joint TE problem

Minimize MLU over all SD demands

Linear program - too large to solve directly

Decompose by SD pair

1 **SD_1: A→B**

Adjust split ratios for this pair only.

Others fixed → *1-D search*

2 **SD_2: A→C**

Adjust split ratios for this pair only.

Others fixed → *1-D search*

3 **SD_3: B→D**

Adjust split ratios for this pair only.

Others fixed → *1-D search*

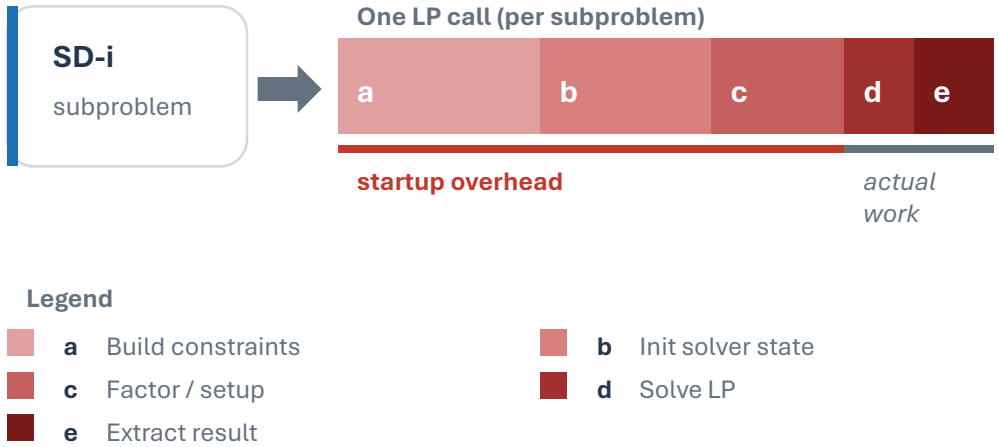
Each query asks whether target u has enough residual path capacity; binary search finds u without an LP solver.*

Solve the SD Subproblem by Searching Target MLU

Replace the LP solver with a binary search over the target MLU.

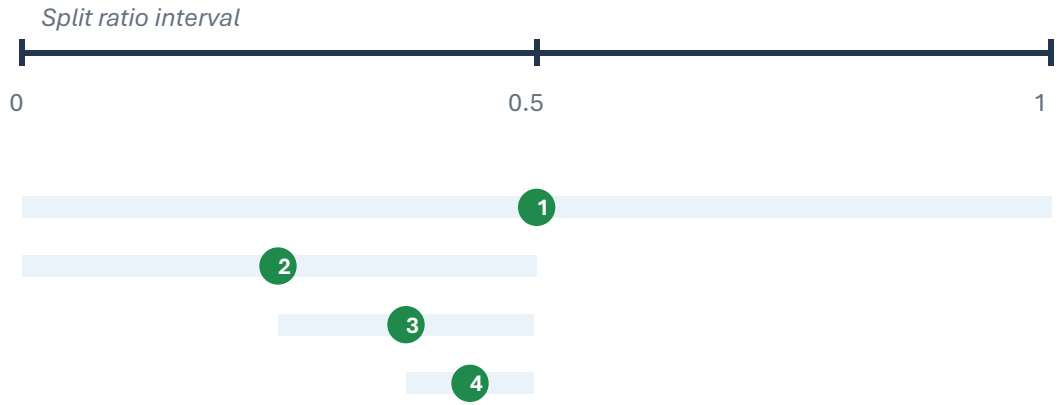
LP solver per subproblem

Even one solver call is heavy — most cost is startup



Binary search over target MLU

1-D subproblem → halve the interval. No solver to start.



Why is binary search valid? Feasibility is monotone in u.

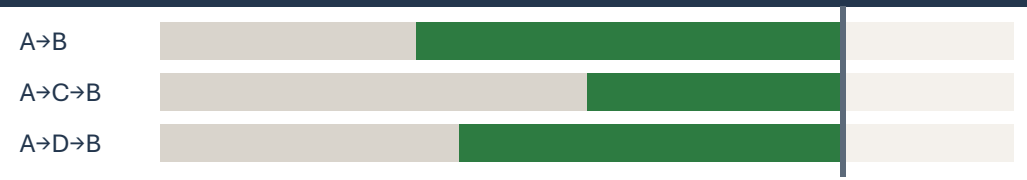
u = 0.5

gray = background traffic | green = demand we try to place | vertical line = cap u

Σ slack < demand → infeasible → raise u

u = 0.8

gray = background traffic | green = demand we try to place | vertical line = cap u



Σ slack ≥ demand → feasible → lower u toward u*

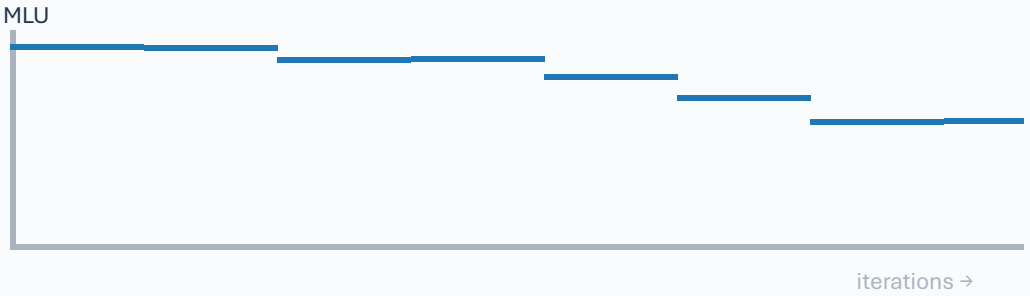
Dynamic Sequence: Focus on the Most-Congested Edges

Each iteration prioritizes SDs whose paths cross the highest-utilization edge — convergence accelerates.

Static round-robin order

Every SD is updated in turn — most do not touch the bottleneck.

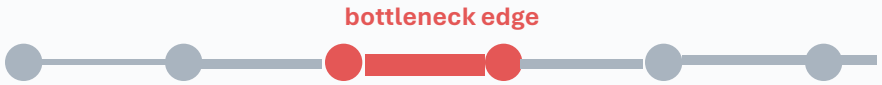
Each SD updated in round-robin



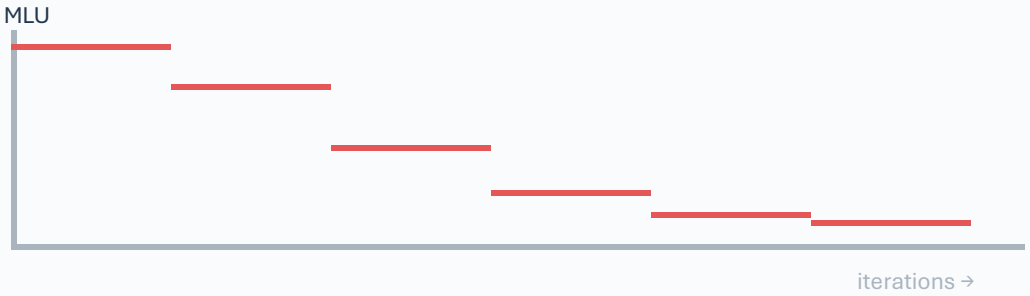
Many iterations spent on SDs that don't affect MLU.

Utilization-driven order

Find the most-congested edge. Update only the SDs that cross it.



Only SDs through the red edge are updated.



Every step hits the bottleneck. Convergence is fast.

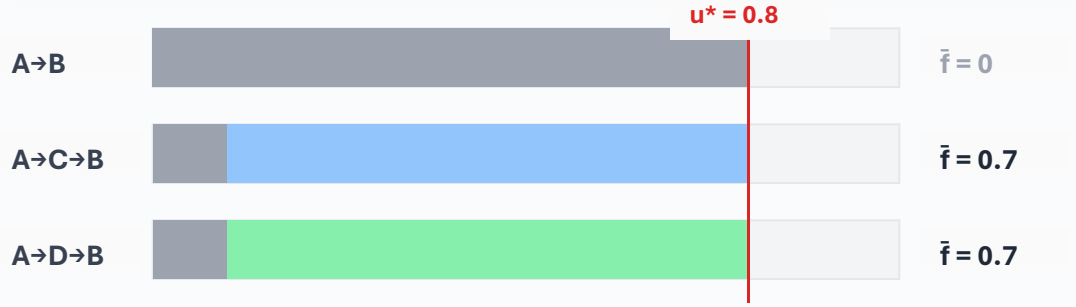
Each iteration prioritizes the bottleneck — fewer wasted updates, faster convergence.

How SSDO Finds the Most Balanced Split

From optimal MLU to balanced split ratios (Figure 4 example: SD = (A, B), u* = 0.8)

1 Background traffic decides u*

at u* = 0.8 the available $\bar{f}$ already overshoots $\Sigma = 1$



$\Sigma \bar{f} = 0 + 0.7 + 0.7 = 1.4 > 1$ → solution exists, but not unique

2 Many feasible splits, same MLU

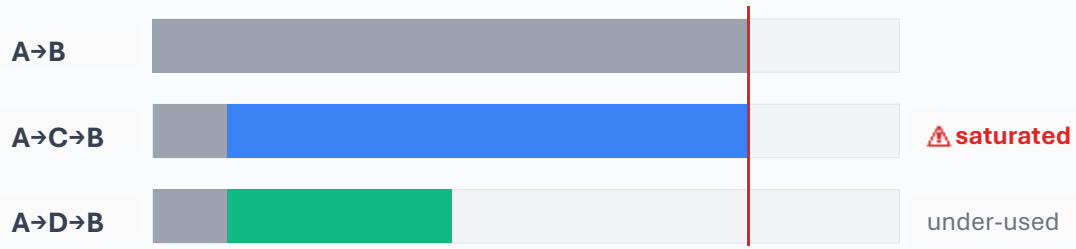
any (f_1, f_2, f_3) with $\Sigma f = 1$ and $f \le \bar{f}$ achieves u* = 0.8

solution	A→B	A→C→B	A→D→B
Sol 1	0.0	0.5	0.5
Sol 2	0.0	0.7	0.3
Sol 3	0.0	0.3	0.7

all three → MLU = u* = 0.8 ❓ which one should SO pick?

3 How to find the most balanced split?

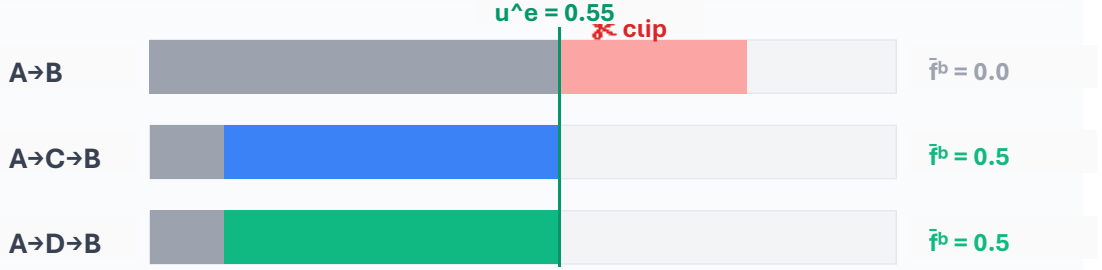
e.g. (0, 0.7, 0.3) saturates A→C→B → blocks future SDs



⚠ imbalanced — we need a tie-breaker among feasible splits

4 SSDO clips & binary-searches u^e

clip negative $\bar{f} \to 0$, then bisect on $\Sigma \bar{f}^b = 1$



$\Sigma \bar{f}^b = 0 + 0.5 + 0.5 = 1$ ✓ unique split (0, 0.5, 0.5)

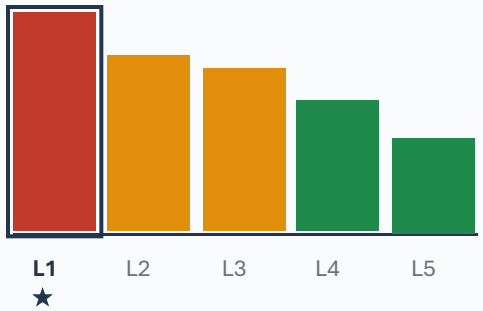
Hottest Edge Shifts Across Iterations

A 4-iteration walk-through on a real topology.

Iteration 1

hot: L1

Top-5 link utilizations



MLU 92%

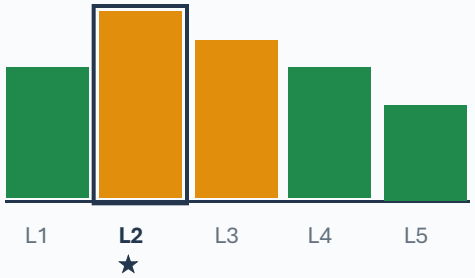
SDs traversing L1:

- SD-A · SD-D · SD-G

Iteration 2

hot: L2

Top-5 link utilizations



MLU 78%

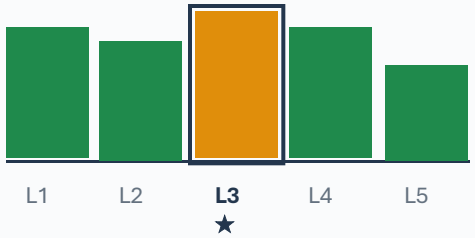
SDs traversing L2:

- SD-B · SD-E

Iteration 3

hot: L3

Top-5 link utilizations



MLU 62%

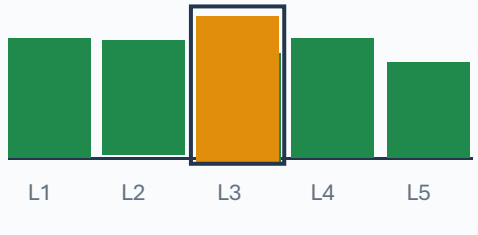
SDs traversing L3:

- SD-C · SD-F · SD-H

Iteration 4

Done

Top-5 link utilizations



MLU 62%

No reduction.

Converged

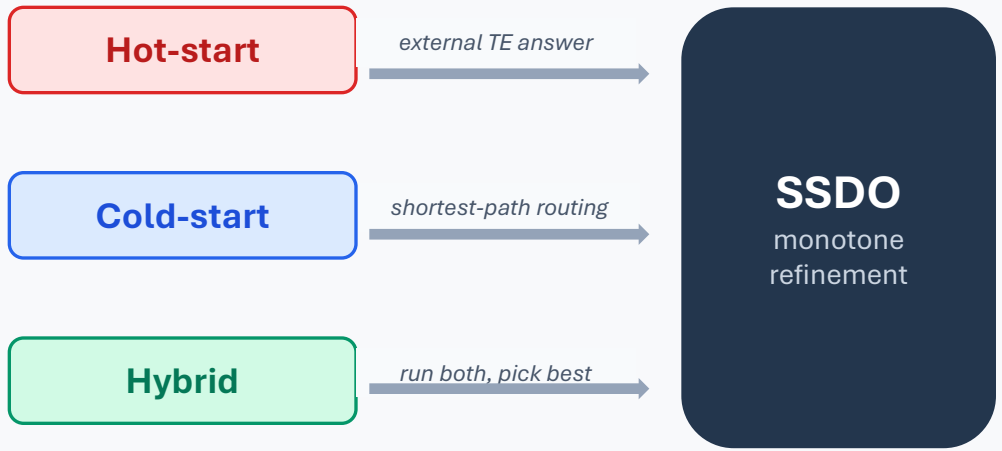
Each iteration optimizes SDs traversing the current hottest link; once it cools down, SSDO moves to the next bottleneck.

SSDO Deployment Strategies

Flexible initialization, anytime stopping, and a path-based form for multi-hop topologies.

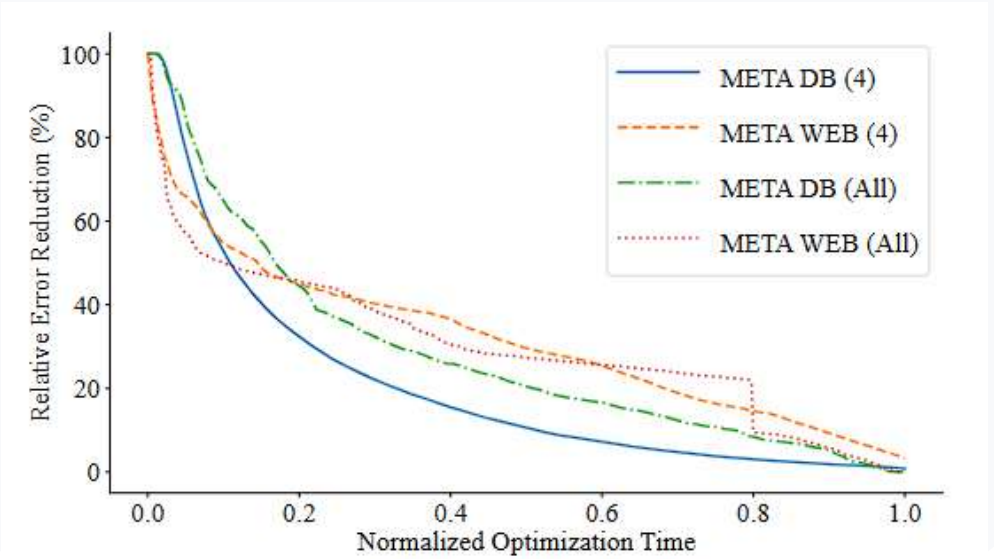
Initialization: hot, cold, or hybrid

Three ways to seed split ratios — all converge into one SSDO engine



Early termination: ship anytime

Error drops fast — stop anytime, keep the best so far



Most of the gain comes early — flexible time–quality trade-off across topologies.

Main Result: Best Quality-Time Tradeoff at Scale

Across Meta topologies, SSDO stays close to LP-all when LP-all is feasible, and keeps solving when several accelerators fail at scale.

Near-optimal when LP-all runs

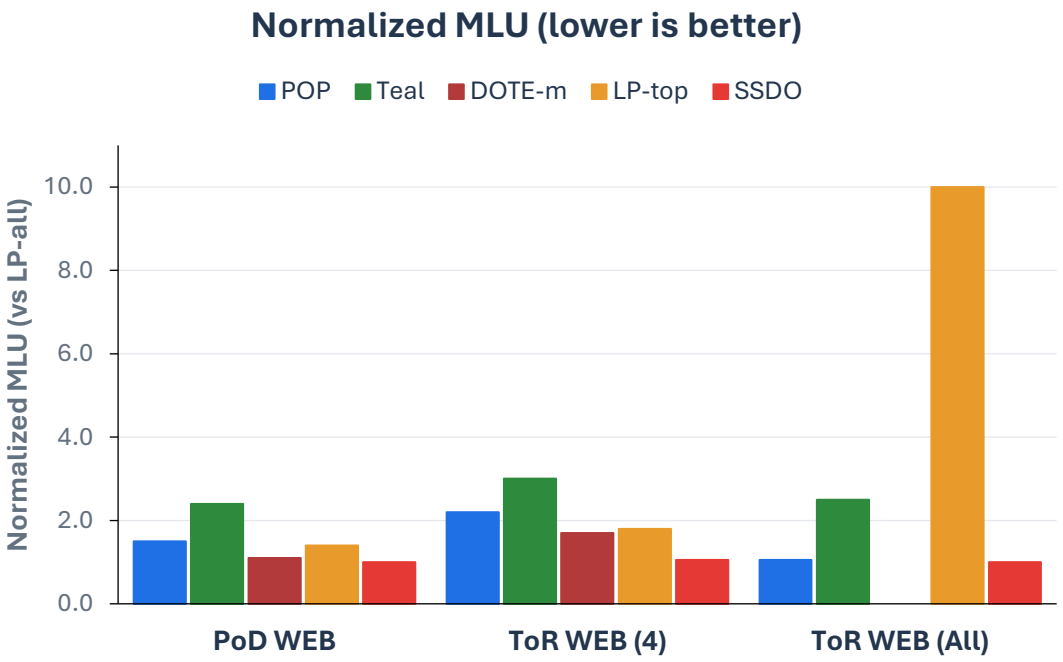
small error vs LP-all on feasible Meta cases

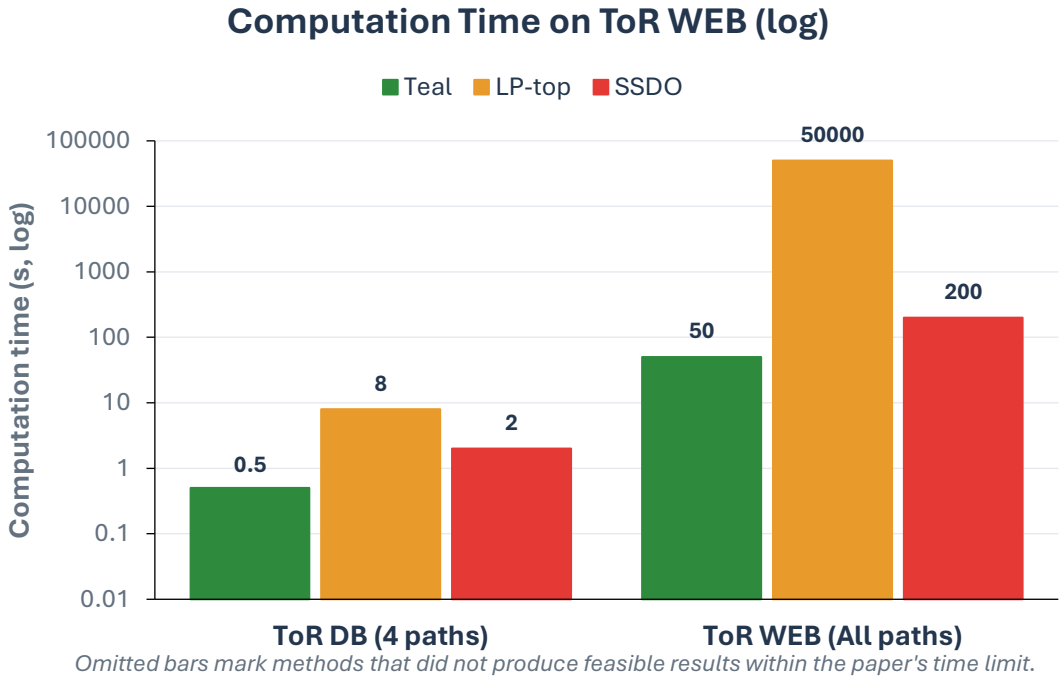
Much faster than direct LP

ToR WEB (4 paths): much lower solve time than LP-all

Handles all-path WEB

SSDO returns a solution where POP and LP-all hit the time limit



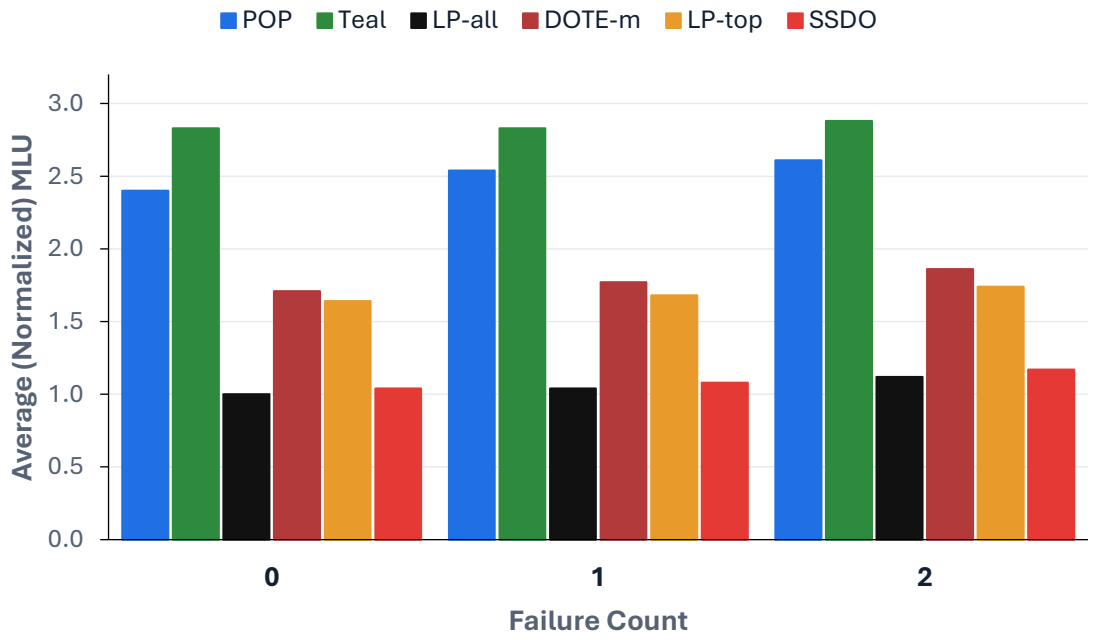


Omitted bars mark methods that did not produce feasible results within the paper's time limit.

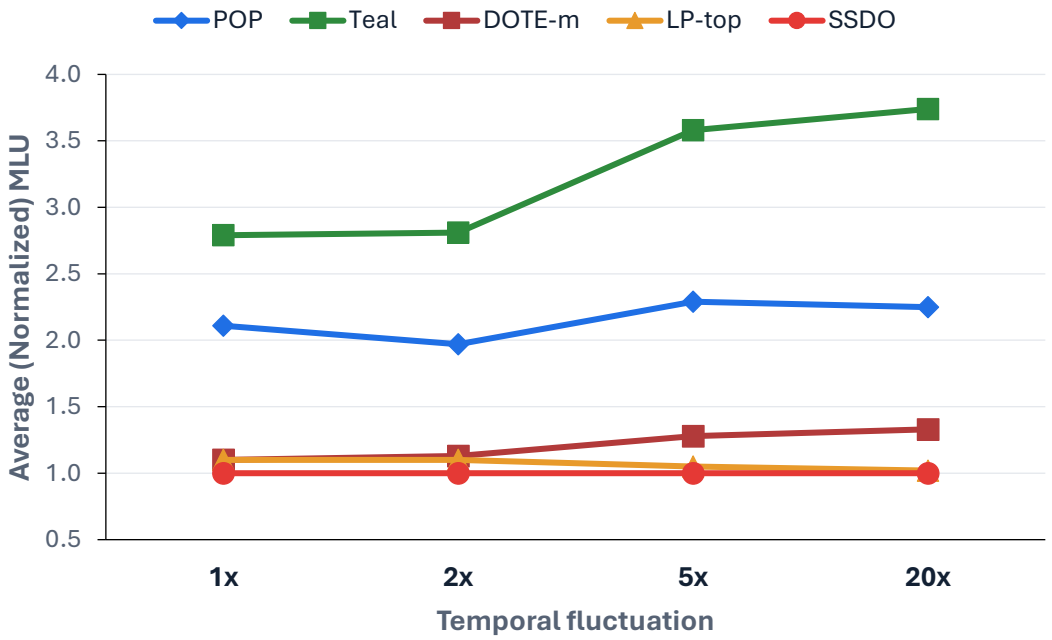
SSDO Remains Strong Under Failures and Demand Shifts

Because SSDO optimizes the current topology and demand directly, it remains reliable under failures and traffic shifts.

Under Link Failures (when operate, failures is known)



Under Demand Fluctuations



Failures

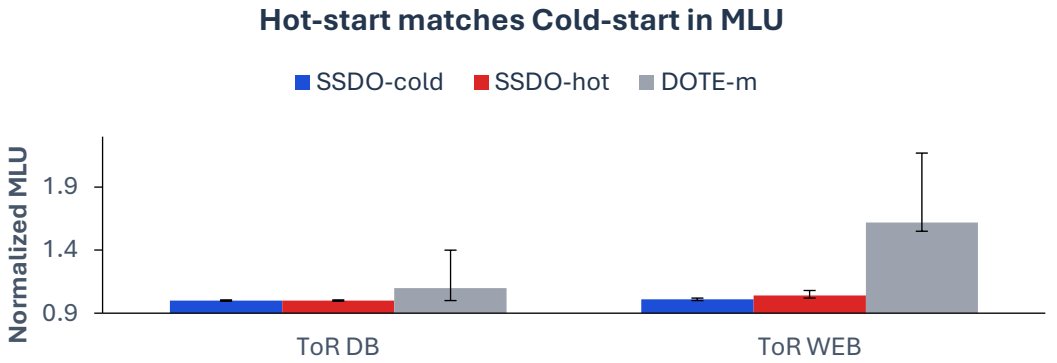
SSDO stays close to LP-all as random link failures increase in Meta WEB (4 paths).

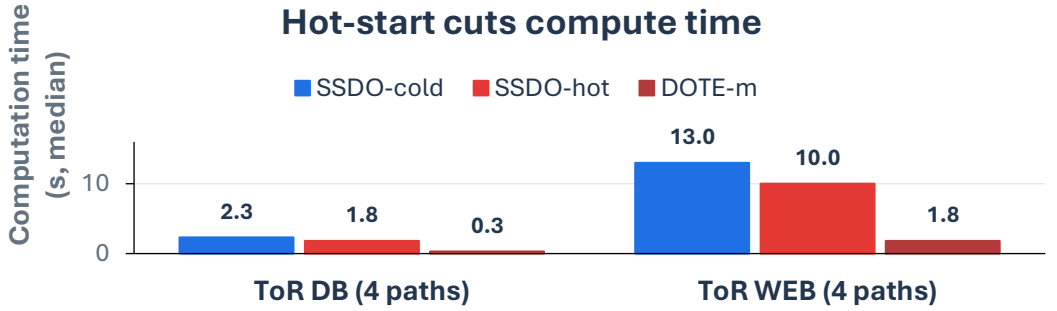
Demand fluctuations

SSDO remains stable as temporal variation increases; learned baselines are more sensitive to distribution shift.

Generality and Anytime Behavior

The path-based formulation also works on WANs, and hot-start/early termination gives useful quality under tight time budgets.





WAN generality
close to LP-all on UsCarrier

Fast convergence
large MLU reductions within seconds

No quality loss
SSDO-hot ≈ SSDO-cold, both far below DOTE-m

And faster
hot-start cuts compute time on both topologies

Thank You

Questions & Discussion

Paper

GitHub
Code & Implementation

arXiv
Paper & Appendices

Homepage
Personal Page

Yingming Mao

Xi'an Jiaotong University/ Shanghai Innovation Institute

mao1234@stu.xjtu.edu.cn