Contracting Wide-area Network Topologies to Solve Flow Problems Quickly: Extended Version

Paper # XXX, 12 pages body, 20 total

Abstract– Many enterprises today manage traffic on their wide-area networks using software-defined traffic engineering schemes, which scale poorly with network size; the solver runtimes and number of forwarding entries needed at switches increase to untenable levels. We describe a novel method, which, instead of solving a multi-commodity flow problem on the network, solves (1) a simpler problem on a contraction of the network, and (2) a set of sub-problems in parallel on disjoint clusters within the network. Our results on the topology and demands from a large enterprise, as well as on publicly available topologies, show that, in the median case, our method nearly matches the solution quality of currently deployed solutions, but is 8× faster and requires 6× fewer FIB entries. We also show the value-add from using a faster solver to track changing demands and to react to faults.

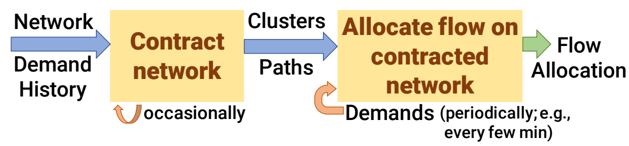


Figure 1: NCFlow's workflow.

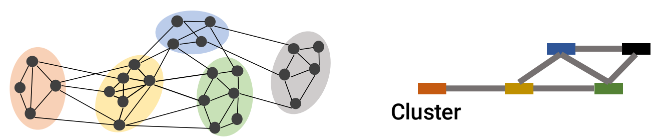


Figure 2: The original network on the left is divided into clusters, shown shaded with different background colors. The contracted network is shown on the right.

1 Introduction

Wide-area networks (WANs), which connect locations across the globe with high-capacity optical fiber, are an expensive resource [7, 35, 36, 38]. Hence, enterprises seek to carefully manage the traffic on their WANs to offer low latency and jitter for customer-facing applications [28, 61, 68] and fast response times for bulk data transfers [45, 56].

The state-of-the-art approach used in several enterprises today [35, 36, 38] is to compute optimal routing schemes for the current demand by solving global multi-commodity flow problems [7,35,36,38]; the global flow problems are re-solved periodically, since demands may change or links may fail, and the computed routes are encoded into switch forwarding tables using software-defined networking techniques [7].

As network sizes grow, solving multi-commodity flow problems on the entire network becomes practically intractable. As noted in [36], the "algorithm run time increased super-linearly with the site count," which led to "extended periods of traffic blackholing during data plane failures, ultimately violating our availability targets," as well as "scaling pressure on limited space in switch forwarding tables." This problem is unlikely to go away: anecdotal reports indicate that WAN footprints today are already over 10× larger than the few tens of sites that were considered in prior work [35, 36], since enterprises have built more sites to move closer to users.

In this paper, we seek to retain the benefits of global traffic management for these larger WAN networks without requiring excessively many forwarding entries at switches or prohibitively long solver runtimes. Also, by using a faster solver, WAN operators can reduce loss when faults occur and carry more traffic on the network by tracking demand changes.

Our solution is motivated by the observation that WAN topologies and demands are *concentrated*: the topology typically has well-connected portions separated by a few, lower-capacity edges, and more demand is between nearby datacenters. This is likely due to multiple operational considerations: (1) submarine cables have become shared choke points for connectivity between continents (see Figure 3), (2) the connectivity over land follows the road or rail networks along which fiber is typically laid out, and (3) enterprises build datacenters close to users, then steer traffic to nearby datacenters [11, 61, 68]. Therefore, more capacity and demand are available between *nearby* nodes, and we analyze data from a large enterprise WAN in §2 to show this.

We leverage this concentration of capacity and demand to decompose the global flow problem into several smaller

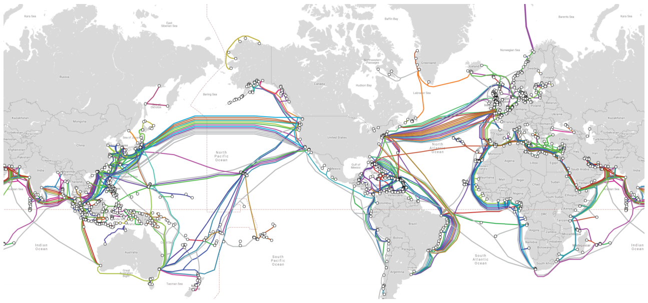


Figure 3: Submarine cables serve as choke points in WAN topologies; figure is excerpted from [62].

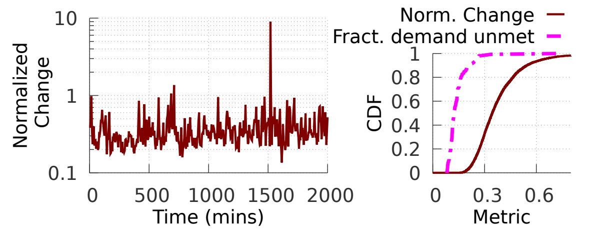


Figure 4: On the left, we plot the L2 norm of the change in the demands between successive 5-minute periods divided by the L2 norm of the traffic matrix at a time. On the right, we show the CDF of this change ratio. We also show a CDF of the fraction of demand that is unsatisfied if using the allocation computed for the previous period.

problems, many of which can be solved in parallel. As shown in Figure 2, we divide the network into multiple connected components, which we refer to as *clusters*. We then solve modified flow problems on each cluster, as well as on the *contracted network*, where nodes are clusters and edges connect clusters that have connected nodes. Prior work [4, 9, 15] notes that Google and other map providers use different contractions to compute shortest paths on road network graphs. Our goal is to closely match the multi-commodity max flow solution in quality (i.e., carry nearly as much total flow), while reducing the solver runtime and number of required forwarding entries. We discuss related work in §7; to our knowledge, we are the first to demonstrate a practical technique for multi-commodity flow problems on large WAN topologies.

Solving flow problems on the contracted network poses two key challenges:

1. How to partition the network into clusters? More clusters leads to greater parallelism, but maximizing the inter-cluster flow requires careful coordination between the sub-problems at multiple clusters.
2. How to design the sub-problems for each cluster to improve speed while reducing inconsistencies in allocation? The sub-problem for a cluster has fewer nodes and edges to consider, but it will not be be faster if it must consider all node pairs whose traffic can pass through the cluster.

Our solution NCFlow[1] achieves a high-quality flow allocation with a low runtime and space complexity by addressing each of these challenges in turn. First, we contract the network using well-studied algorithms such as modularity-based clustering [25] and spectral clustering [53], which are designed to identify the choke-point edges in a network. Second, we *bundle* demands whose sources and/or targets are in the same cluster, treating them as a single demand. In Figure 2 for example, the yellow cluster considers as one bundled demand all traffic from source nodes in the red cluster to target nodes in the green cluster. Doing so can lead to inconsistent flow allocations between clusters (which we explain in §3.1.1) and we devise careful heuristics to provably avoid them (§3.2). Finally, we reduce the forwarding entries needed at switches by reusing pathlets within clusters and traffic splitting rules across multiple demands (§3.5).

Figure 1 shows the workflow for NCFlow. First, we choose appropriate clusters and paths using an offline procedure over historical traffic—these choices are pushed into the switch forwarding entries. This step happens infrequently, such as when the topology and/or traffic changes substantially. Then, online (e.g., once every few minutes), NCFlow computes how best to route the traffic over the clusters and paths, similar to deployed solutions [35, 36, 38].

Overall, our key contributions are:

- We propose NCFlow, a decomposition of the multi-commodity max flow problem into an offline clustering step and an online, provably feasible, algorithm that solves a set of smaller sub-problems in parallel.
- We evaluate NCFlow using real traffic on a large enterprise WAN, as well as synthetic traffic on eleven topologies from the Internet Topology Zoo [6]. Our results show that, for multi-commodity max flow, NCFlow is within 2% of the total flow allocated by state-of-the-art path-based LP solvers [35, 36, 38] in 50% of cases; NCFlow is within 20% in 97% of cases. Furthermore, NCFlow is at least $8\times$ faster than path-based LP solvers in the median case; in 20% of cases, NCFlow is over $30\times$ faster. Lastly, NCFlow requires 2.7–16.7$\times$ fewer forwarding entries in the evaluated topologies. NCFlow also compares favorably to state-of-the-art approximation algorithms [27, 41] and oblivious techniques [43, 57].
- We show that, as a fast approximate solver, NCFlow can be used to react quickly to demand changes and link failures. Specifically, in comparison to TEAVAR [19], NCFlow carries more flow when no faults occur and suffers about the same amount of total loss during failures.

We have open-sourced an anonymous version of NCFlow [2], and are in the early stages of integrating NCFlow into production use at a large enterprise.

2 Background and Motivation

We analyze the changes in topology and traffic on a large enterprise WAN over a several-month period. As Figure 4

[1]short for **N**etwork **C**ontractions for **Flow** problems

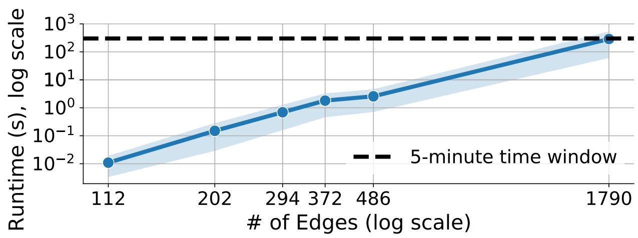


Figure 5: Runtimes of a state-of-the-art solver on topologies from Internet Topology Zoo [6]. Both axes are in log scale and the band represents standard deviation. In production WANs, new traffic demands arrive every few minutes [35, 38].

shows, the change in traffic demand from one 5-minute window to the next is substantial; the average change is 35%; in 20% of the cases, the traffic change is over 45%. The enterprise solves a global flow allocation problem every few minutes. The figure on the right shows the fraction of traffic that will remain unsatisfied if the flow allocation from the previous window were to be used instead of computing a new allocation. We see that the median loss is 13%; in 20% of the cases, over 20% of the demand remains unsatisfied. We verify that computing a new allocation will satisfy all of the demand; using the previous window's allocation causes loss because some datacenter pairs may receive more flow in the previous allocation than their current demand while other datacenter pairs go unsatisfied.

Given the above data, computing a new allocation in each time window is needed to carry more traffic on the WAN. However, solver runtime increases super-linearly with the size of the topology, as shown in Figure 5. For several public topologies and on a variety of traffic matrices, we benchmark the multi-commodity max-flow problem (specifically PF_4, as will be described in §5.1). The runtimes were measured on a server-grade machine using a production-grade optimization library [33]. As the figure shows, when the topology size exceeds a thousand edges, the time to compute a flow allocation can exceed the allotted time window.

A fast solver would not only ensure that new allocations complete in time—it could also enable more frequent allocations, e.g., every minute. Doing so would enable allocations to track changing demands at a finer granularity. Moreover, as we show in §5, a fast solver can help when reacting to link and switch failures.

We ground our observation on the concentration of demand and capacity by measuring data from the production WAN. Our measurements reveal the following:

Demand properties:

- On average, 7% (or 16%) of the node pairs account for half (or 75%) of the total demand.
- When nodes are divided into a few tens of clusters, 47% of the total traffic stays within clusters. If the demands were distributed uniformly across node pairs, only 8% of the traffic would stay within clusters; thus the demand within clusters is about 6× larger than would be expected from a uniform distribution.

WAN topology properties:

- When nodes are divided into tens of clusters, 76% of all edges and 87% of total capacity is within clusters.
- The skew in capacity is small: the ratio between the largest edge capacity and the mean is 10.4.
- The skew in node degree is also small: the average node degree is 3.9, with $\sigma = 2.6$; the max is 16.
- The diameter of the network is small (=11) and the average shortest-path length is also small (= 5.3) relative to the network size.

Motivated by the above analyses, NCFlow seeks to be a fast solver for large WAN topologies by leveraging the concentration of traffic demands and capacity.

Background: Before we describe NCFlow's design, we give some background on multi-commodity flow problems. Given a set of nodes, capacitated edges, and demands between nodes, a flow allocation is feasible if it satisfies demand and capacity constraints. The goal of a multi-commodity flow problem is to find a feasible flow which optimizes a given objective; Table 1 lists some common flavors.

The fastest algorithms [27, 41] are approximate; i.e., given a parameter ε, they achieve at least $(1-\varepsilon)\times$ the optimal value. However, their runtime complexity is at least quadratic (see Table 1).

Moreover, these solutions allow demands to travel on any edge, thus requiring millions of forwarding table entries at each switch for thousand-node topologies. Thus, production systems [35, 38] restrict flow to a small number of preconfigured paths per demand, which reduces the required forwarding table entries by 10–100×.

Using notation from Table 2, the feasible flow over a preconfigured set of paths can be defined as:

$$\begin{aligned}
&\mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \{f_k \mid \forall k \in \mathcal{D} \text{ and} && (1)\\
&f_k = \sum_{p\in\mathcal{P}_k} f_k^p, && \forall k \in \mathcal{D} \quad \text{(flow for demand } k)\\
&f_k \le d_k, && \forall k \in \mathcal{D} \quad \text{(flow below volume)}\\
&\sum_{\forall k, p\in\mathcal{P}_k, e\in p} f_k^p \le c_e, && \forall e \in \mathcal{E} \quad \text{(flow below capacity)}\\
&f_k^p \ge 0 && \forall p \in \mathcal{P}, k \in \mathcal{D} \quad \text{(non-negative flow)} \ \}
\end{aligned}$$

Production systems use linear optimization-based solvers [35, 36, 38]. On WANs with thousands of nodes, the optimization problem could have millions of variables and equations just to verify that a flow allocation is feasible.

In this paper, we consider the problem of maximizing the total flow across all demands:

$$\begin{aligned}
&\mathsf{MaxFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} f_k && (2)\\
&\text{s.t.} \quad \mathbf{f} \in \mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P})
\end{aligned}$$

	Maximization term	Additional Constraints	Used in	Known best complexity
MaxFlow	$\sum_{k\in\mathcal{D}} f_k$	none	[35, 38]	$O(M^2\epsilon^{-2}\log^{O(1)} M)$ [27]
MaxFlow with Cost Budget	$\sum_{k\in\mathcal{D}} f_k$	$\sum_k \sum_{p\in\mathcal{P}_k} \sum_{e\in p} f_k^p \mathsf{Cost}_e \le \mathsf{Budget}$		$O(\epsilon^{-2} M \log M (M + N\log N) \log^{O(1)} M)$ [27]
Max Concurrent Flow	α	$d_k\alpha \le f_k, \forall k \in \mathcal{D}$	[19, 39, 40]	$O(\epsilon^{-2}(M^2+KN)\log^{O(1)} M)$ [41]

Table 1: We illustrate a few different multi-commodity flow problems all of which find feasible flows but optimize for different objectives and can have additional constraints; see notation in Table 2. More problems are discussed in [10].

Term	Meaning
$\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}$	Sets of nodes, edges, demands, and paths
N, M, K	The numbers of nodes, edges, and demands, i.e., $N = \lvert\mathcal{V}\rvert, M = \lvert\mathcal{E}\rvert, K = \lvert\mathcal{D}\rvert$
e, c_e, p	Edge e has capacity c_e; path p is a set of connected edges
(s_k, t_k, d_k)	Each demand k in $\mathcal{D}$ has source and target nodes ($s_k, t_k \in \mathcal{V}$) and a non-negative volume (d_k).
$\mathbf{f}, f_k^p$	Flow assignment vector for a set of demands and the flow for demand k on path p.

Table 2: Notation for framing multi-commodity flow problems.

$\mathcal{V}_{\text{agg}}$, $\mathcal{E}_{\text{agg}}$, $\mathcal{D}_{\text{agg}}$, $\mathcal{P}_{\text{agg}}$	Nodes, edges, demands, and paths in the aggregated graph
$\mathcal{V}_x, \mathcal{E}_x, \mathcal{D}_x, \mathcal{P}_x$	Subscript denotes entities in the restricted graph for cluster x
x, η	Each cluster x is a strongly connected set of nodes and η is the number of clusters

Table 3: Additional notation specific to NCFlow.

MaxAggFlow

$\mathbf{f_1} \triangleq \mathsf{MaxFlow}(\mathcal{V}_{\text{agg}}, \mathcal{E}_{\text{agg}}, \mathcal{D}_{\text{agg}}, \mathcal{P}_{\text{agg}})$

MaxClusterFlow

$\forall$clusters $x, \mathbf{f_2^x} \triangleq \mathsf{MaxFlow}(\mathcal{V}_x, \mathcal{E}_x, \mathcal{D}_x, \mathcal{P}_x)$

s.t. $\mathsf{NoMoreFlowThruCluster}(\mathbf{f}, \mathbf{f_1}, x, \mathcal{P}_{\text{agg}})$ (see §B.3)

MinPathE2E

$\mathbf{f_3} \triangleq \{f_k, \forall k \in \mathcal{D}_{\text{agg}}\}$ s.t.

s.t. $\mathsf{NoMoreAlongPath}(\mathbf{f}, \mathbf{f_2}, \mathcal{P}_{\text{agg}})$ (see §B.3)

SrcTargetMax

$\forall$clusters $x, y, x \neq y$, $\quad \mathbf{f_4^{xy}} \triangleq \arg\max \sum_{k\in K_{xy}} f_k$

$$s.t. \sum_{k\in K_{sy}} f_k \le f^x_{2,K_{sy}}, \quad \forall s \in x; \qquad \sum_{k\in K_{xt}} f_k \le f^y_{2,K_{xt}}, \quad \forall t \in y;$$

$$\sum_{k\in K_{xy}} f_k \le f_{3,K_{xy}}; \qquad f_k \le d_k, \qquad \forall k \in K_{xy}$$

Figure 6: The basic flow allocation algorithm used by NCFlow; notation used here is defined in Table 3.

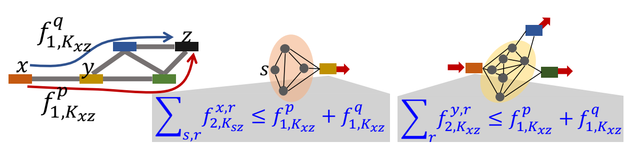


Figure 7: An example illustrating how the flow allocated in MaxAggFlow translates to constraints on the flow to be allocated in MaxClusterFlow at two different clusters.

3 NCFlow

In this section, we describe NCFlow. Our steps are as shown in Figure 1. Offline, based on historical demands, we divide the network into clusters and determine paths. Further details are in §3.4. Online, we allocate flow to the current demands by solving a carefully constructed set of simpler sub-problems, some of which can be solved independently and in parallel. We describe these sub-problems in §3.1. Although they can be solved quickly, disagreements between independent solutions can lead to infeasible allocations; we present a simple heuristic in §3.2 that provably leads to feasible flow allocations. In §3.3, we discuss extensions that increase the total flow allocated by NCFlow. We also show sufficient conditions under which NCFlow is optimal and matches the flow allocated by MaxFlow. Finally, in §3.5, we discuss how NCFlow uses fewer forwarding entries by reusing pathlets within clusters and splitting rules for different demands.

3.1 Basic Flow Allocation

We begin by describing a simple (but incomplete) version of NCFlow's flow allocation algorithm; the pseudocode is in Figure 6. We continue using Figure 2 as a running example. The basic algorithm proceeds in four steps.

In the first step, we allocate flow on the aggregated graph; as shown in MaxAggFlow in Figure 6. In the aggregated graph, an example of which is in Figure 2 (right), nodes are clusters and the edges are bundled edges from the original graph—the edge between the red and yellow clusters corresponds to the five edges between these clusters on the actual graph. Similarly, we bundle demands on the aggregated graph: the demand K_{xy} between the clusters x and y corresponds to all of the demands whose sources are in cluster x and targets are in cluster y. The resulting flow allocation ($\mathbf{f_1}$) accounts for bottlenecks on the edges between clusters. However, this flow may not be feasible, since there may be bottlenecks *within* the clusters.

In the second step, we refine the allocation from step 1 to account for intra-cluster demands and constraints. Specifically, we allocate flow for the demands whose sources and targets are within the cluster. We also allocate no more flow than was allocated in $\mathbf{f_1}$ for the inter-cluster flows. MaxClusterFlow in Figure 6 shows code for this step. We note a few details:

- We use virtual nodes to act as the sources and targets for the inter-cluster flows; the flow allocated in $\mathbf{f_1}$ determines which virtual node (i.e., which neighboring cluster) is the sender or the receiver for an inter-cluster demand.
- Figure 7 shows two examples on the right where the virtual nodes are drawn using squares.
- Figure 7 also shows the NoMoreFlowThruCluster constraints for demands from sources in the red cluster to

Problem	# of Nodes	# of Edges	# of Demands
MaxFlow	N	M	K
MaxAggFlow	η	$\leq \min(M, \eta^2)$	$\leq \min(K, \eta^2)$
MaxClusterFlow	$\sim \frac{N}{\eta} + \eta$	$\sim \frac{M}{\eta} + 2\eta$	$\sim \frac{K}{\eta^2} + 2\frac{N}{\eta} + \eta^2$

Table 4: Sizes of the problems in Figure 6; notation is described in Tables 2 and 3. Just verifying that flow is feasible (i.e., FeasibleFlow in Eq. 1) uses $O(\#\text{ nodes} * \#\text{ edges})$ number of equations and variables. NCFlow has one instance of MaxAggFlow and executes the η instances of MaxClusterFlow in parallel. The other two problems, MinPathE2E and SrcTargetMax, are relatively insignificant.

targets in the black cluster (depicted as x and z respectively). On the aggregated graph, the flow for this demand takes the two paths shown. In the red cluster, as shown in the equation, the traffic from all sources (s), along multiple paths (r), to the virtual node is restricted to be no more than what was allocated in $\mathbf{f_1}$.

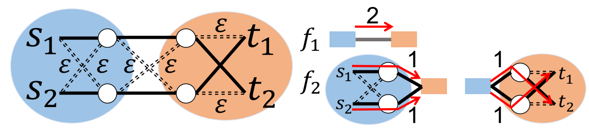
- Figure 7 on the right also shows a more complex case that happens in the yellow cluster. Here, the traffic arrives from one virtual node but can leave to multiple virtual nodes. In MaxClusterFlow, we set up paths between all pairs of virtual nodes. As shown in the equation, the traffic leaving the red virtual node on paths (r) to either of the other virtual nodes must be no more than the total flow on paths p and q from $\mathbf{f_1}$.
- Observe that bundling demands ensures fewer variables and constraints for MaxClusterFlow. The demand from red to black clusters comprises twenty node pairs in the actual graph in Figure 2 (left); four sources in the red cluster and five targets in the black cluster. However, the MaxClusterFlow for the red cluster only has four bundled demands, from each source to the virtual node, and the yellow cluster has just one bundled demand from and to virtual nodes.

In the third step, we reconcile end-to-end; that is, we find the largest flow that can be carried along each path on the aggregate graph. As shown by MinPathE2E in Figure 6, for each bundle of demands and each path, we take the minimum flow allocated ($\mathbf{f_2^x}$) at each cluster on the path.

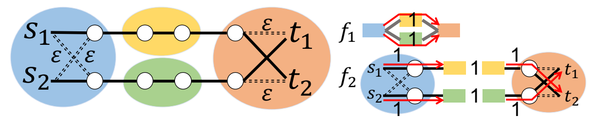
The flow allocation for the demands in a cluster x can be read directly from the $\mathbf{f_2^x}$ solution of MaxClusterFlow. For demands that span clusters, however, more work remains because the steps thus far do not directly compute their flow. In particular, $\mathbf{f_3}$ allocates flow for cluster bundles; such as say for all the demands whose sources are in cluster x and targets are in cluster y. At the same time, the per-cluster flow allocations $\mathbf{f_2^x}$ and $\mathbf{f_2^y}$ allocate flow from a source node and to a given target respectively. Thus, in the final step, SrcTargetMax, we assign the maximal flow to each inter-cluster demand that respects all previous allocations.

3.1.1 Properties of Basic Flow Allocation

Solver runtime: The numbers of equations and variables in the sub-problems are shown in Table 4. If the number of clusters η is 1, note that there is exactly one per-cluster problem, MaxClusterFlow, which matches the original problem from Eqn. 2. When using a few tens of clusters, we will show in §5 that all of the sub-problems are substantially smaller than the original problem (MaxFlow).



(a) Disagreement arising from bundling edges: As shown on the right, the basic flow allocation in Figure 6 will compute a flow of 2 units, but only 4ε units of flow can be carried; see §3.1.1.



(b) Disagreement arising from bundling demands: As shown on the right, the basic flow allocation in Figure 6 will again compute a flow of 2 units, but only 4ε units of flow can be carried.

Figure 8: Illustrating how disagreements in flow allocation can occur in the basic flow allocation algorithm.

Feasibility: The flow allocated by Figure 6 satisfies demand and capacity constraints; we will prove this formally in §A.1. For demands whose source and target are in different clusters, however, disagreements may ensue since the different problem instances assign flow to different bundles of edges and demands. We illustrate two such examples in Figure 8; both have 1 unit of demand from s_1 to t_1 and from s_2 to t_2. The dashed edges have a capacity of $\varepsilon \ll 1$ and all of the other edges have a very large capacity.

- The example in Figure 8a illustrates an issue with bundling edges. The actual graph on the left can only carry 2ε units of flow for each demand. However, as the figures on the right show, MaxAggFlow allocates two units of flow since the four edges between these two clusters can together carry all of the two units of demand. The MaxClusterFlow instances also allocate two units of flow as shown. The discrepancy arises because the problems in Figure 6 do not know that the *top* egress of the left cluster can take in all of the demand of s_1 but has only a low capacity to t_1.
- The example in Figure 8b illustrates an issue with bundling demands. Here too, observing the actual network on the left will show that 2ε units can be carried for each demand split evenly between the top and the bottom path. Again, as the figures on the right show, the basic flow allocation algorithm will conclude that both units of demand can be carried. Here, the discrepancy arises from the bundling of demands, the problems in Figure 6 do not realize that the MaxClusterFlow instance of the left cluster wants to send the first demand to the brown cluster while the MaxClusterFlow of the right cluster wants to receive the second demand from the brown cluster.

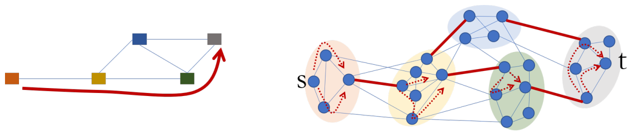


Figure 9: To guarantee feasibility, each cluster bundle is allocated flow on only one path on the aggregated graph (left) and on only one edge between each pair of clusters (right); the usable path and edges are shown in dark red. Note that multiple paths can still be used within clusters.

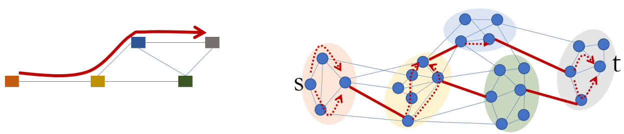


Figure 10: Contrasting with Figure 9, for the same cluster bundle, in a subsequent iteration, NCFlow allocates flow on a different path on the aggregate graph and on different inter-cluster edges. The chosen paths and edges are again shown in red.

3.2 A feasible heuristic

To avoid end-to-end disagreements, we make two simple changes to the basic flow allocation in §3.1.

First, when solving MaxAggFlow, only one path on the aggregated graph can be used for all of the demands between a given pair of clusters; we call such groups of demands to be cluster bundles. Next, between a pair of connected clusters, only one edge can carry the flow for a cluster bundle. Figure 9 shows in dark red an example path for a cluster bundle and the allowed edges between clusters; we also show the intra-cluster paths that can carry flow for this bundle.

There are multiple ways to avoid disagreements while keeping the problem sizes small via bundling; we discuss more options in [12]. We discuss the above changes here because they are simple and sufficient. Specifically, we show that:

Theorem 1. *The algorithm in Figure 6, when constrained as discussed above, will always output a feasible flow.*

Proof. The proof is in §A.2. Intuitively, these changes suffice because the independent decisions made by different problems in Figure 6 cannot disagree; per cluster bundle, all problem instances allocate flow to the same edge and path. □

3.3 Stepping towards optimality

It is easy to see that the flow allocation algorithm described thus far is fast but not optimal; that is, it may allocate less total flow over all demands than the flow allocated by solving the larger global problem (MaxFlow from Eqn. 2). There are a few reasons why this happens. The MaxAggFlow in Figure 6 allocates flow on paths through clusters without knowing how much flow the clusters can carry. Switching the order, i.e., solving MaxClusterFlow before MaxAggFlow, could be worse because each cluster must allocate flow without knowing how much flow can be carried end-to-end. Furthermore, the heuristic in §3.2 constrains a cluster bundle from using multiple edges between clusters and from using multiple paths on the aggregated graph. Here, we discuss a few extensions to increase the total flow allocated by NCFlow.

First, we re-solve the problems in Figure 6 multiple times. That is, after each execution, we deduct allocated flow and use the (smaller) residual capacity on edges in the next iteration. Also, we pick different edges between clusters and/or different paths on the aggregated graph in different iterations (see Figure 10 for an example). The number of iterations is configurable; we continue as long as the total flow increases in each iteration by at least a pre-specified amount (say 5%). One could apply other policies as well such as a timeout. We show in §5 that a small number of iterations result in a sizable increase in the total flow. We will also show that later iterations finish faster than the first iteration perhaps because fewer demands remain to satisfy.

Next, we empirically observe that the choice of clusters and edges/paths to use in different iterations has a sizable effect on flow allocation. For instance, the disagreements in Figure 8 *go away* by using a different choice of clusters—specifically, place all nodes on the top of each network graph into one cluster and the rest into a different cluster. We discuss how NCFlow precomputes cluster and edge/path choice in §3.4.

To wrap up, we prove that flow allocation will be optimal when a few sufficient conditions hold:

Theorem 2. *The method in Figure 6 leads to the optimal flow allocation when the number of clusters is* 1 *or equal to the number of nodes or all of the following conditions hold:*

- *the aggregated graph $\mathcal{G}_{agg}$ is a tree,*
- *only one edge connects any pair of clusters,*
- *any number of paths can be used within a cluster and,*
- *all demands are satisfiable.*

Proof. By optimal, we mean that the total allocated flow must be as large as that of the global MaxFlow which can use any number of paths. The proof is in §A.3. Intuitively, when the number of clusters is 1, a single instance of MaxClusterFlow is identical to the global MaxFlow problem. When the number of clusters equals the number of nodes, MaxAggFlow is identical to MaxFlow. Furthermore, the conditions listed lead to optimality because the optimal flow allocation can be transformed into an allocation that can be outputted by Figure 6. □

Even though the listed conditions appear restrictive, note that the topology within clusters can be arbitrary. We will show in §5 that NCFlow offers nearly optimal flow allocations even when the above conditions do not hold.

3.4 Clustering and choosing paths

The choice of clusters crucially affects both the solution quality and runtime of NCFlow. We compute a set of clusters and choose edges/paths in an offline manner. Graph partitioning is well-studied in the literature [5, 21, 64]; the following aspects of the partitioning problem are pertinent to NCFlow:

- **Concentrated:** NCFlow can output better flow allocations when a large fraction of the total demand is between nodes in the same cluster and the total capacity of edges within clusters is larger than that of the edges between clusters.
- **Many, even:** Intuitively, NCFlow will have a smaller runtime by balancing the complexity of MaxAggFlow with that of MaxClusterFlow. As the number of clusters increases from 1 to N, the runtime of the former increases while that of the latter decreases. If clusters are equal in size, the optimal number of clusters is roughly $\sqrt{N}$. Notably, reducing the size and degree of the largest cluster (which gives rise to the most time-consuming instance of MaxClusterFlow) often reduces runtime.

NCFlow sweeps the following clustering strategies for several values of the number of clusters and picks the clusters that offer the best outcome for a topology on historical demands.

- **Modularity:** Prior algorithms [18, 25] divide nodes into clusters by maximizing a "modularity" scoring function over the edges within and between clusters. In NCFlow, we apply modularity-based clustering by using the edge's capacity as its weight during execution.
- **Spectral clustering:** Using the same weights as above, we use a recent non-linear technique [53] to compute eigenvectors of the weighted adjacency matrix and choose as many of the top eigenvectors as the desired number of clusters; each node is assigned to the cluster of their closest eigenvector (e.g., using k-means).

Path choice in NCFlow: On the aggregated graph and on each cluster graph, we pre-compute offline a small number of paths between every pair of nodes. We consider the following different path choices and pick paths that lead to the largest flow allocation on historical demands:

- k-shortest paths [69] with edge weight of 1 or $\frac{1}{c_e}$ where c_e is the capacity of edge e.
- As above, but with the additional requirement that the paths for a node pair are edge-disjoint [52].

NCFlow also pre-computes offline (1) a pseudo-random choice of which edges to use between a pair of connected clusters in each iteration and (2) which path on the aggregated graph to use for each cluster bundled demand in each iteration. We explore a few choices such as uniformly random, weighted by capacity, and balancing the assigned historical demands.

We do not know if these clustering techniques and path choices are best suited for the problem at hand. Our claim is that NCFlow offers good results with these choices; we expect future work will consider more techniques [5, 42, 64].

3.5 Setting up switch forwarding entries

NCFlow uses many fewer switch forwarding entries than prior works due to the following reasons.

First, the paths along which NCFlow allocates flow can be thought of as a sequence of pathlets [32, 46, 67] in each cluster connected by crossing edges between clusters. Figures 9 and 10 illustrate such paths on the right. This observation is crucial because a pathlet can be reused by multiple demands. For example, in Figure 9, the flow from any source in the red cluster to any target in the grey cluster would use the same pathlets shown in the yellow, green, and blue clusters. Prior work [35, 36], on the other hand, establishes paths for each demand. Using pathlets has two advantages. The number of pathlets used by NCFlow is about η times less than the number of paths used by prior works[2]. Furthermore, a typical pathlet has fewer hops than a typical end-to-end path. Thus, NCFlow uses many fewer rules to encode paths in switches.

Next, whenever NCFlow allocates flow at the granularity of cluster bundled demands, all of the demands in a bundle take the same paths and are split in the same way across paths. Hence, NCFlow reuses the same traffic splitting rule for all demands in such bundles. For instance, the demands from source s in the red cluster in Figure 9 to any target in the grey cluster are split with the same ratio across the same pathlets in all clusters (except the grey cluster where they take different pathlets to reach their different targets). Thus, NCFlow uses substantially fewer splitting rules in switches. For instance, the number of splitting rules at a source decreases by a factor of $\sqrt{N}/2$[3].

The paths and splitting rules to push into switch forwarding tables are determined by the offline component of NCFlow and only change occasionally. After each allocation, only the splitting ratios change. More details on the data-plane of NCFlow such as how to compute the total flow that can be sent by each demand and the splitting ratios as well as how to move packets from one pathlet to the next are in Appendix B. In §5, we measure the numbers of rules used by NCFlow.

4 Implementing NCFlow

Our current prototype of NCFlow is about 5K lines of Python code, which invokes Gurobi [33] v8.1.1 to solve all of the optimization problems. For clustering WAN topologies, we adapt [26] to find clusters that maximize modularity; we also use our own implementation of NJW spectral clustering [53]. We use a grid search over the number of clusters (η) and the

[2] More precisely, the number reduces from $PN(N-1)$ to $\sum_x P(N_x)(N_x-1)$ where P is the number of paths per node pair, the actual graph has N nodes divided into η clusters and cluster x has N_x nodes. If nodes are divided evenly into the clusters, $N_x = N/\eta$, and the ratio of these terms is $\sim \eta$.

[3] A source requires $N-1$ splitting rules in prior works but with NCFlow only requires $N_x - 1 + \eta - 1$ where N_x is the number of nodes in the source's cluster; assuming nodes are divided evenly into clusters and that $\eta \sim \sqrt{N}$, the ratio of these terms is $\sqrt{N}/2$.

Topology	# Nodes	# Edges	# Clusters
PrivateLarge	$\sim$ 1000s	$\sim$ 1000s	31
Kdl	754	1790	81
PrivateSmall	$\sim$ 100s	$\sim$ 1000s	42
Cogentco	197	486	42
UsCarrier	158	378	36
Colt	153	354	36
GtsCe	149	386	36
TataNld	145	372	36
DialtelecomCz	138	302	33
Ion	125	292	33
Deltacom	113	322	30
Interoute	110	294	20
Uninett2010	74	202	24

Table 5: Some of the WAN topologies used in our evaluation; see §5.1.

above clustering techniques to identify the best performing choice for each topology on a set of historical traffic matrices. The grid search also identifies which path selection parameters to use. To compare with state-of-the-art techniques, we customize the public implementations of SMORE [43, 44] and TEAVAR [19]. We have also implemented Fleischer's algorithm [27]; our implementation is about 10$\times$ faster than public implementations of related algorithms [8, 37]. These code artefacts are available on GitHub [2].

5 Evaluation

We evaluate NCFlow on several WAN topologies, traffic matrices, and failure scenarios to answer the following questions:

- Compared to state-of-the-art LP solvers and approximate combinatorial algorithms, does NCFlow offer a good trade-off between runtime and total flow allocation? Is it substantially faster, with only a small decrease in total flow?
- For real-world TE scenarios, in which flow solvers must adapt to changing demands and faults, how much benefit does NCFlow offer relative to the state-of-art?
- How do our various design choices in NCFlow impact its performance?

5.1 Methodology

Here, we describe our methodology—the topologies, traffic, baselines, and metrics used in our evaluation.

Topologies: We use two real topologies from a large enterprise—PrivateSmall is a production internet-facing WAN with hundreds of sites, and PrivateLarge is a larger WAN that contains many more sites. We also use several topologies from the Internet Topology Zoo [6] and reuse topologies used by prior works [19, 38]. Table 5 shows details for some of the used topologies; note that the topologies shown are $10--100\times$ larger than those considered by prior work [19, 35, 38, 43, 48].

Traffic Matrices (TMs): We benchmark NCFlow on traffic traces from PrivateSmall, which contain the total traffic between node pairs at 5-minute intervals. We also generate the following kinds of synthetic traffic matrices for all topologies:

- **Gravity**(v) [14, 60]: The total traffic leaving a node is proportional to the total capacity on the node's outgoing links (parameterized by v); this traffic is divided among other nodes proportional to the total capacity on their incoming links.
- **Uniform**$([0,a))$: The traffic between any pair of nodes is chosen uniformly at random, between 0 and a.
- **Bimodal**$([0,a),[b,c),p)$ [14]: A p fraction of the node pairs, chosen uniformly at random, receive demands from **Uniform**$[b,c)$ while the rest receive demands from **Uniform**$[0,a)$. We use $p = 0.2$.
- **Poisson**(λ,δ): The demand between nodes s and t is a Poisson random variable with mean $\lambda\delta^{\mathrm{d}_{st}}$, where d_{st} is the hop length of the shortest path between s and t and $\delta \in [0,1)$ is a *decay factor*. We choose δ close to 0 or to 1 to model strongly and weakly concentrated demands, respectively.

For each model above, we select parameters such that fully satisfying the traffic matrix leads to a maximum link utilization of about 10% in each topology. Then, we scale all entries in the TM by a constant $\alpha \in \{1,2,4,8,16,32,64,128\}$. Doing so creates demands that range from easily satisfiable to only partially satisfiable; with $\alpha = 128$, the satisfiable portion of the demand varies between 25-70%. We generate five samples for each traffic model and scale factor for each topology.

Baselines: We compare NCFlow with these techniques:

Path Formulation (PF_4) solves the multi-commodity max-flow problem shown in Equation 2 using k-shortest paths between node pairs where $k = 4$.

PF *Warm Start* ($\mathsf{PF}_{4\mathsf{w}}$) matches PF_4 except that it allows the LP solver to "warm start"; that is, over a sequence of traffic matrices, the flow allocated to the previous TM is used as a starting point to compute allocation for the next TM. When traffic changes are small, warm start leads to faster solutions.

Approximate Combinatorial Algorithms: Fleischer's algorithm [27] is the best-known approximation algorithm for MaxFlow as discussed in §2. We use two variants: Fleischer-Path where flow is restricted to a path set and Fleischer-Edge without any path restrictions. We show results here for an approximation guarantee of 0.5; that is, the techniques must achieve at least half of the optimal flow allocation. Results for other approximate values are in [12].

SMORE [43] allocates flow dynamically on paths that are pre-computed using Räcke's Randomized Routing Trees (RRTs). We use the code from [44] to compute paths. Since the LP in [44] requires demands to be fully satisfiable, we implement a variant, SMORE*, that maximizes the total flow on the computed paths, regardless of demand satisfiability.

TEAVAR [3,19] models link failure probabilities and computes flow allocations given an availability target. We implement a variant, TEAVAR*, that maximizes the total flow[4]; further details are in Appendix C.

Clusters, Paths, and # of Iterations: Table 5 shows the number of clusters used by NCFlow per topology. As noted in §3.4 and §4, we use an offline grid search to identify clusters that perform well on historical demands. Here, we report results on edge-disjoint paths, chosen using inverse capacity as the edge length; results for other path choices are qualitatively similar (see [12]). All schemes that use paths (i.e., PF_4, Fleischer-Path, TEAVAR*, and NCFlow) use the same method to compute paths. For each iteration up to $I = 6$, we also pre-compute offline (1) which path to use on the aggregated graph, and (2) which edge to use between connected clusters (see §3.2).

Metrics: We compare the baselines on the following metrics:

- **Relative total flow** is the total flow achieved by a scheme relative to PF_4.
- **Speedup ratio** is the runtime of each scheme relative to PF_4. For LP-based methods, we report the Gurobi solver runtimes, since models can be constructed once offline in practice. For combinatorial methods, we report algorithm execution time. All runtimes are measured on an Intel Xeon 2.3GHz CPU (E52673v4) with 16 cores and 112 GB of RAM.
- **FIB Entries:** We measure the number of switch forwarding entries used.

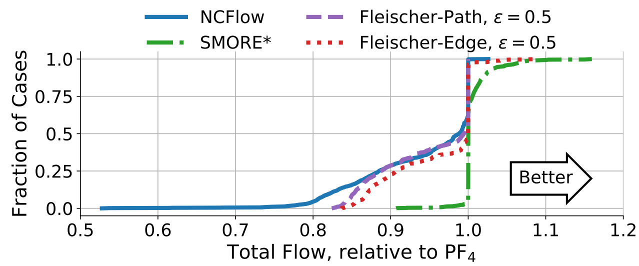
5.2 Comparing NCFlow to the State of the Art

Figures 11a and 11b show cumulative density functions (CDFs) of the relative total flow and speedup ratio for NCFlow and several baselines. These results consist of 2,600 traffic matrices and 13 topologies. If a scheme matches the baseline PF_4, its CDF will be a pulse at $x = 1$ in both figures; the fraction of cases to the left (or right) of $x = 1$ indicate how often a scheme is worse (or better) than PF_4. Note that the x-axis for the speedup ratio is in log scale.

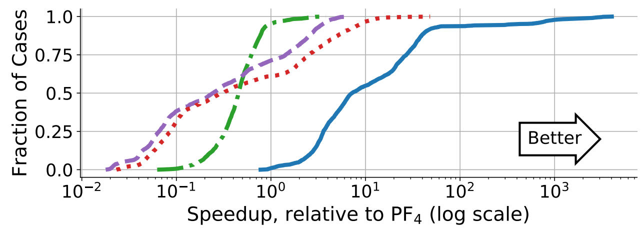
We see that SMORE*, shown using green dashed lines in the figures, modestly improves the flow allocation (in 25% of the cases) while almost always taking longer to run than PF_4. Both effects are because SMORE* allocates flow on Räcke's RRTs instead of k-shortest paths.

The edge and path variants of Fleischer's, shown using purple and red lines in the figures, perform similarly; since they are approximate algorithms, they allocate less flow than PF_4 in roughly 50% of cases, but are also faster than PF_4 in slightly less than 50% of cases. We conclude that these approximate algorithms are not practically better than the baseline, PF_4.

[4]TEAVAR [3,19] maximizes the *concurrent* flow; see Table 1



(a) CDF of total flow relative to PF_4



(b) CDF of speedup relative to PF_4

Figure 11: CDFs comparing NCFlow with state-of-the-art methods. With only a modest decrease in total flow, NCFlow offers a substantial runtime speedup.

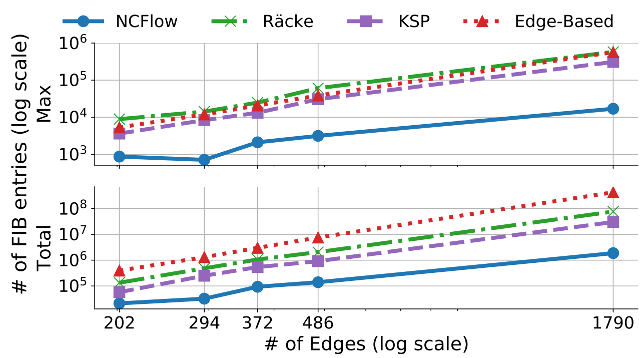


Figure 12: Comparing the number of forwarding entries used by various methods for the experiments from Figure 11.

In contrast, NCFlow, shown with dark blue lines in the figures, almost always allocates at least 80% of PF_4's total flow, while achieving large speedups. In the median case, NCFlow achieves 98% of the flow and is over 8× faster. These improvements accrue from NCFlow solving smaller optimization problems than PF_4.

For the same experiments considered above, Figure 12 shows the number of switch forwarding entries used in different topologies. (A full set of results is in Table 6 in the Appendix.) The bottom plot is the total number of forwarding entries across all switches, while the top shows the maximum for any switch. Note that the shared x-axis is in log scale. NCFlow consistently uses fewer forwarding entries; using NCFlow offers a greater amount of relative savings than switching from all edges to just a handful of paths per demand. The savings from NCFlow also increase with topology size. The reason, as noted in §3.5, is that NCFlow reuses pathlets and traffic splitting rules for many different demands.

To further understand the performance of NCFlow, Fig-

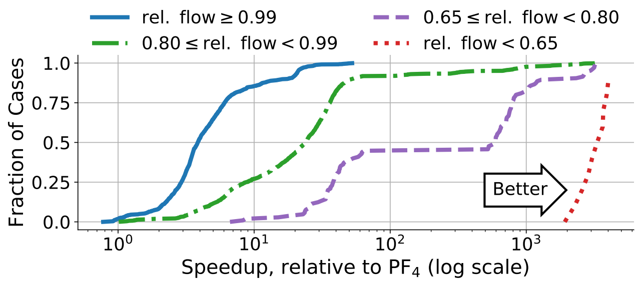


Figure 13: Breaking down the NCFlow results from Figure 11b into four separate CDFs based on relative total flow.

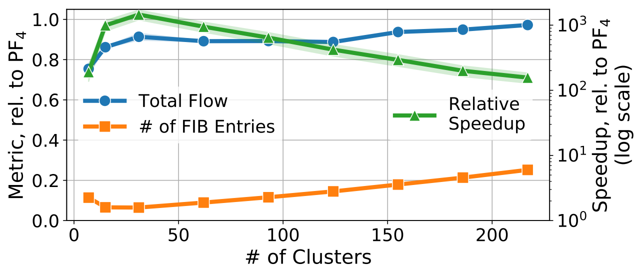


Figure 14: NCFlow's performance when using different numbers of clusters on PrivateLarge. The speedup ratio is plotted on the right y-axis in log scale; the other metrics use the left y-axis.

ure 13 breaks down the above experiments into four ranges based on total relative flow. We plot CDFs of the speedup ratio per range. The solid blue and green dashed line, which correspond to relative flow above 0.99 and in $[0.8, 0.99)$ respectively, account for 49% and 46% of all experiments. The figure shows that NCFlow achieves sizable speedups while allocating large amounts of flow. Figure 19 presents similar breakdowns along various dimensions of interest, such as the topology size, amount of traffic, and traffic model. In sum, we find that NCFlow improves across the board and excels on large topologies with concentrated demands.

5.3 Effect of Design Choices

Recall from §3.3 that NCFlow uses multiple iterations of Figure 6. In the above experiments, NCFlow allocates on average over 90% of flow during the first iteration. The first iteration alone accounts for over 75% of the runtime. Later iterations are faster, because they have less traffic to consider and are primarily useful when the first iteration under-allocates flow.

Breaking down the runtime of NCFlow by the steps in Figure 6 identifies several cases where MaxClusterFlow alone accounts for over 70% of NCFlow's runtime, because the largest cluster contains a large fraction of the nodes. Exploring more clustering techniques or recursively using NCFlow to subdivide the largest clusters can lead to further improvements.

Figure 14 shows how NCFlow's performance varies with the numbers of clusters used on PrivateLarge. While NCFlow allocates roughly the same amount of total flow, using about 30 clusters improves runtime and reduces forwarding entries; for more details, see [12].

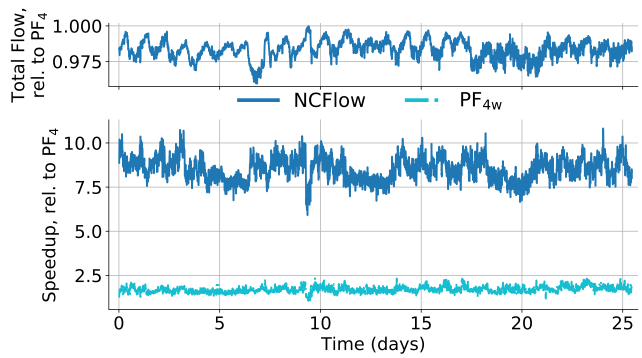


Figure 15: Allocated flow and speedup relative to PF_4 on a sequence of production TMs from PrivateSmall. In half of the cases, NCFlow allocates at least 98.5% of the flow and is at least 8.5× faster.

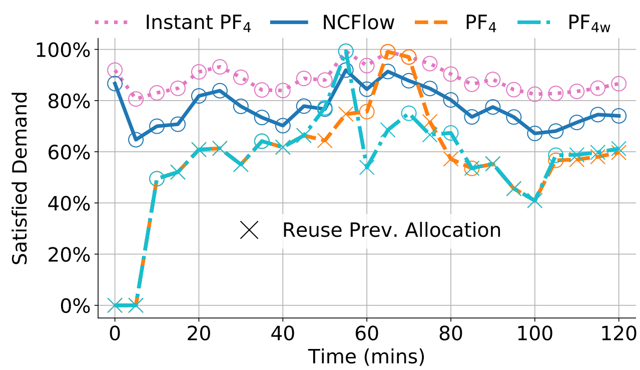


Figure 16: The effect of solver runtime on the ability to keep track of changing demands on PrivateLarge. PF_4's and PF_{4w}'s slow runtimes cause them to carry 62% of the traffic that can be satisfied by Instant PF_4, a (hypothetical) scheme which has zero runtime. NCFlow's faster runtime compensates for its sub-optimality; it satisfies 87% of Instant PF_4's total flow.

5.4 NCFlow on Real-World Traffic

Here, we experiment with a sequence of traffic traces collected on the PrivateSmall WAN. Figure 15 plots the moving average (over 5 windows) of the total flow and speedup relative to PF_4 for two schemes—NCFlow in blue and PF_{4w} in light blue. The figure shows that PF_{4w}'s warm start yields a median speedup of 1.66×. NCFlow achieves a consistently higher speedup (8.5× in the median case), and the flow allocation is nearly optimal: the median total relative flow is 98.5%, and NCFlow always allocates more than 93%.

5.5 Tracking Changing Demands

Here, we evaluate the impact of a technique's runtime on its ability to stay on track when demands change. Specifically, on the PrivateLarge topology, we use a time-series of traffic matrices, wherein a new TM arrives every five minutes and the change from one TM to the next is consistent with the findings in Figure 4[5]. At each time-step, all techniques have the opportunity to compute a new allocation for the current TM or to continue computing the allocation for an earlier

[5]For more details, see Figure 20

TM if they have not yet finished; in the latter case, their most recently computed allocation will be used for the current TM. For example, a technique that requires five minutes to compute a new allocation will be always *one window behind*, i.e., each TM will receive the allocation that was computed for the previous TM.

Figure 16 shows the fraction of demand that is satisfied by three different schemes; we also show the value for an instantaneous scheme which is not penalized for its runtime. We find that PF_4's average runtime is over 15 minutes; hence, as the orange dashed line shows, PF_4 is able to compute a new allocation only for every third or fourth TM. This leads to substantial demand being unsatisfied: for node pairs whose current demand is larger than before, PF_4's will not allocate enough flow. On the other hand, node pairs whose current demand is less than their earlier demand will be unable to fully use PF_4's allocation. As the figure shows, PF_4 only satisfies 53% of the demand on average, whereas Instant PF_4 satisfies 87% of the demand. Thus, the inability to track changing demands causes PF_4 to carry almost 40% less flow.

PF_{4w} (the dash-dot light blue line) is modestly faster than PF_4 on average, but has more variance—in some cases, it is much faster. As the figure shows, the average demand satisfied by PF_{4w} is only slightly larger than PF_4—about 54%.

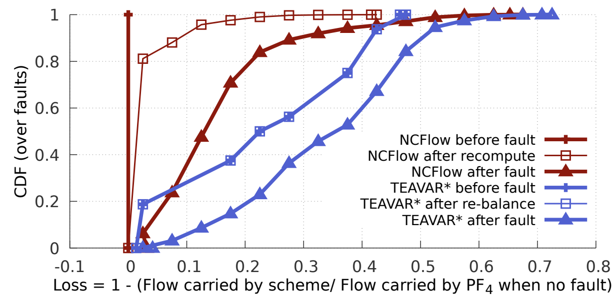
In contrast, NCFlow (the solid dark blue line) does not allocate the maximum possible flow to any TM, but finishes well within five minutes, allowing it to keep track of the changing demands. We find that on average NCFlow satisfies 75% of the demands; its smaller runtime more than makes up for sub-optimality, allowing NCFlow to carry more flow than PF_4 when demands change.

5.6 Handling Failures with NCFlow

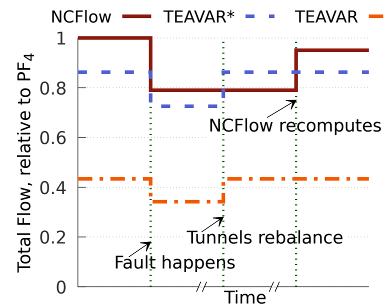
Here, we evaluate the effect of link failures. As we note in §C, TEAVAR* did not finish within several days on any of the topologies listed in Table 5: when all possible 2-link failure scenarios are considered, the number of equations and variables in the optimization problem increases from $O(N^2)$ for MaxFlow to $O(M^2N^2)$ for TEAVAR [19], where N and M are the numbers of nodes and edges, respectively. Hence, we report results on the 12-node, 38-edge WAN topology from B4 [38]. We generate synthetic traffic matrices as noted in §5.1. Using link failure probabilities from TEAVAR [3], we generate several hundred failure scenarios and, for each TM, we measure the flow carried by NCFlow and TEAVAR* before the fault, immediately after the fault, and after recovery.

A key difference in the fault recovery models between NCFlow and TEAVAR* is that TEAVAR* requires source nodes to rebalance the traffic splits when a failure happens; doing so takes at least one round-trip time (RTT) on the WAN. Given a parameter β, TEAVAR* guarantees that there will be no flow loss after the tunnels re-balance with a probability of $1-\beta$[6].

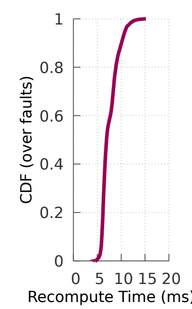
[6]This is imprecise, see §C which also has further results.



(a) CDFs of the flow *loss* before faults, immediately after faults and after recovery (B4 topology, many traffic matrices and faults; see §5.6).



(b) Timelapse of when a fault occurs (B4 topology, Uniform traffic matrix, $\beta = 0.99$)



(c) NCFlow's time to recompute after fault.

Figure 17: Comparing failure response of NCFlow with prior work.

We ran TEAVAR* with $\beta = 0.99$, as recommended in [19]. NCFlow, on the other hand, recomputes flow allocations taking into account the links that have failed; doing so takes one execution of NCFlow and a few RTT to change the traffic split ratios at switches[7]. Figure 17c shows a CDF of the solver runtime used by NCFlow to recompute; we see that this time is well within one RTT on the WAN.

Figure 17b shows a timelapse of the flow carried on the network before the fault, immediately after the fault, and after recovery. As the figure shows, TEAVAR* is likely to have a smaller loss and for a shorter duration; i.e., until sources rebalance tunnels. NCFlow can carry more flow before fault and after recovery; moreover, the fast solver time can reduce the duration of loss.

Figure 17a shows CDFs over many faults and traffic matrices for NCFlow and TEAVAR*. We record the flow loss at three stages: before the fault, immediately after the fault, and after recovery. As the figure shows, NCFlow's ability to carry more flow before the fault and after recovery more than compensates for the slightly larger loss it may accrue in between.

6 Discussion

NCFlow is **agnostic to the underlying solver** and can use any fast solver for the sub-problems in Figure 6. Thus, NCFlow will benefit from future improvements to LP solvers and approximate methods [27, 30, 41] which can be extended to accommodate NCFlow's additional constraints.

[7]More details on our fault model are in §D.

Further use cases: Beyond serving as a drop-in replacement for today's production WAN traffic controllers, NCFlow can be used in a few other places where fast and close-to-optimal solutions are desirable such as: when allocating flow for future time-steps [39, 40] or to compare the usefulness of different topology changes [1, 22] or to accelerate the training of ML-based routing systems [63].

Extensions and guarantees: Applying NCFlow to other multi-commodity flow problems is important future work. Together, the basic algorithm in §3.1 and the heuristic in §3.2 guarantee feasibility, and FeasibleFlow is a common constraint across several problems (see Table 1). More work is needed, however, to support additional objectives and improve solution quality. When computing maximum total flow, in our experiments, choosing appropriate clusters consistently leads to a small optimality gap. We illustrate some hand-crafted adversarial examples in [12]. Obtaining a guarantee on the total flow allocated by NCFlow remains future work.

7 Related Work

NCFlow builds upon a few themes in prior work. We discuss and evaluate against some prior works already. To recap: (1) Some large enterprises use path-based global optimization problems similar to MaxFlow to manage traffic on their WANs [35, 36, 38]. We saw in §5 that doing so does not scale to the WAN topologies of today or the future, which consist of thousands of sites; (2) We saw that approximate algorithms for multi-commodity max flow, such as [27], require a large number of switch forwarding entries since they can send flow along any edge. Also, NCFlow allocates more flow and is faster compared to path-based versions of these algorithms. (3) Probabilistic fault protection schemes such as TEAVAR [19] take infeasibly long to run on large topologies when considering multiple link failures; they also allocate less flow to reserve capacity to deal with possible failures. Other oblivious techniques [13, 14, 19, 43, 48, 65] have a similar tradeoff. Quickly recomputing using NCFlow trades off slightly more loss after a fault to carry much more traffic before the fault and after recomputation; hence, we believe that NCFlow is better suited to enterprise WANs, which target very high link utilization and have traffic that is elastic to short-term loss (e.g., scavenger-class traffic, such as replicating large datasets [35, 38, 48]). Here, we discuss other related work.

TE on WANs: Typically, a WAN node is not a single switch, but rather a group of switches connected in a specific way such as a full mesh. Similarly, a WAN edge is a systematic collection of links between many different switches. [36] discusses how to hide the intra-node connectivity from the global TE solution. NCFlow complements this technique; it can use a similar intra-node scheme and can support WANs that are 10× larger than were considered in [36]. The specific contraction used by NCFlow—node clusters with large capacity and/or demand between themselves—also differs from the contractions used in route planning [4, 9, 15]. Some BGP-based TE schemes [24, 61, 68], which address how best to move traffic between different (BGP) domains, are also complementary to NCFlow which considers the WAN of a single enterprise (domain). A few other traffic engineering schemes [29, 39, 45, 51] use different protocols, such as DNS and OSPF, or work over timescales of hours to days.

Multi-Commodity Flow Solutions: Both the edge- and path-based LP formulations are well-studied in the literature [16, 66]. Some works consider the case of a single commodity, i.e., one source and one target, and do not directly extend to the case of multiple commodities [34, 47, 55]. The best-known approximate algorithms for multi-commodity flow problems incrementally allocate flow on the shortest path and increase the potential value of all edges on that path [17, 27, 30, 41]. For the problem sizes considered here, we find LP solvers such as Gurobi to be faster in practice, perhaps because they can take larger steps towards the optimal allocation. A few works customize LP solvers to improve performance on flow problems [23, 49]. Note that NCFlow is agnostic to the solver used; that is, NCFlow can use any fast solver for the sub-problems in Figure 6.

Decomposition techniques such as Dantzig-Wolfe and Benders [16, 20] solve large optimization problems by merging the solutions of sub-problems. A few works have used such decompositions for multi-commodity flow problems with inconclusive results [31, 54]; they are not consistently faster than MaxFlow. NCFlow can be thought of as a problem-specific decomposition that leverages the observation that both capacity and demands are concentrated in today's WANs—prior work uses the same observation to improve topology design [50].

8 Conclusion

We present a fast and practical solution for allocating flow on large WANs. We leverage the concentrated nature of demands and topologies to divide nodes into clusters and solve sub-problems per cluster and on the aggregated graph. Our heuristics guarantee feasibility and empirically achieve close-to-optimal flow allocations. By reusing pathlets and splitting rules across demands, we require fewer forwarding entries in switches. Empirically, on topologies that are over 10× larger than were considered in prior work and many traffic matrices, our solution NCFlow is 8.2× faster than the state of the art, while allocating 98.8% of the total flow and using 6× fewer forwarding entries in the median case. We demonstrate that NCFlow offers sizable benefits when tracking changing demands and reacting to failures. As enterprise WANs continue to grow, we believe techniques such as NCFlow can enable improved traffic orchestration and higher link utilization.

References

[1] Capacity planning for the Google backbone network. https://bit.ly/2lViJ4t.

[2] Code for NCFlow and Baselines. https://github.com/netcontract/ncflow.

[3] Code for TEAVAR. https://github.com/manyaghobadi/teavar.

[4] Contraction Hierarchies Path Finding Algorithm. https://bit.ly/3eaiqtg.

[5] GAP: Generalizable Approximate Graph Partitioning Framework. https://arxiv.org/pdf/1903.00614.pdf.

[6] Internet Topology Zoo. http://www.topology-zoo.org/.

[7] Market Trends: SD-WAN and NFV for Enterprise Network Services. https://gtnr.it/3c8hNyA.

[8] Cristinel Ababei. Code for Karakostas. https://bit.ly/2woSloP.

[9] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *Experimental Algorithms*, 2011.

[10] Ravindra Ahuja, Thomas Magnanti, and James Orlin. *Network Flows. Theory, Algorithms, and Applications*. Prentice Hall.

[11] Muthukaruppan Annamalai et al. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *OSDI*, 2018.

[12] Anon. Solving Flow Problems Quickly by Contracting Wide-area Network Topologies: Extended Version. https://bit.ly/2RYit22.

[13] D. Applegate, L. Breslau, and E. Cohen. Coping with Network Failures: Routing Strategies for Optimal Demand Oblivious Restoration. In *SIGMETRICS*, 2004.

[14] David Applegate and Edith Cohen. Making Intra-Domain Routing Robust to Changing and Uncertain Traffic Demands. In *SIGCOMM*, 2003.

[15] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. *CoRR*, 2015.

[16] Dimitris Bertsimas and John N Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena Scientific Belmont, MA, 1997.

[17] Daniel Bienstock. *Potential function methods for approximately solving linear programming problems: theory and practice*, volume 53. Springer Science & Business Media, 2002.

[18] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks, 2008.

[19] Jeremy Bogle et al. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *SIGCOMM*, 2019.

[20] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[21] P. Brucker. On the complexity of clustering problems. In *Optimizing and Operations Research*, Berlin, West Germany, 1977. Springer-Verlag.

[22] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. Robust validation of network designs under uncertain demands and failures. In *NSDI*, 2017.

[23] P. Chardaire and A. Lisser. Simplex and Interior Point Specialized Algorithms for Solving Nonoriented Multicommodity Flow Problems. *Operations Research*, 2002.

[24] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, et al. Taiji: Managing Global User Traffic for Large-Scale Internet Services at the Edge. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 430–446, 2019.

[25] A. Clauset, M.E.J. Newman, and C. Moore. Finding community structure in very large networks. *Phys. Rev.*, 2004.

[26] Aaron Clauset. Fast Modularity Community Structure Inference Algorithm. https://bit.ly/3aAVGQH.

[27] Lisa K. Fleischer. Approximating Fractional Multicommodity Flow Independent of the Number of Commodities. *SIAM J. Discret. Math.*, 2000.

[28] Ken Florance. How Netflix Works With ISPs Around the Globe to Deliver a Great Viewing Experience. https://bit.ly/2RYYrEM, 2016.

[29] B. Fortz and Mikkel Thorup. Internet Traffic Engineering by Optimizing OSPF Weights in a Changing World. In *INFOCOM*, 2000.

[30] Naveen Garg and Jochen Könemann. Faster and Simpler Algorithms for Multicommodity Flow and Fractional Packing Problems. *SIAM J. Comput.*, 2007.

[31] A. M. Geoffrion and G. W. Graves. Multicommodity Distribution System Design by Benders Decomposition. *Management Science*, 1974.

[32] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. In *SIGCOMM*, 2009.

[33] Zonghao Gu, Edward Rothberg, and Robert Bixby. Gurobi optimizer reference manual, version 5.0. *Gurobi Optimization Inc., Houston, USA*, 2012.

[34] Jeff Hartline and Alexa Sharp. Hierarchical Flow. Technical Report 2004-09-29, Cornell University, 2004.

[35] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[36] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In *SIGCOMM*, 2018.

[37] Yuanfang Hu, Yi Zhu, Hongyu Chen, Ronald L. Graham, and Chung-Kuan Cheng. Communication latency aware low power NoC synthesis. In *DAC*, 2006.

[38] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, and Min Zhu. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[39] Virajith Jalaparti, Ivan Bliznets, Srikanth Kandula, Brendan Lucier, and Ishai Menache. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. In *SIGCOMM*, 2016.

[40] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for Wide Area Networks. In *SIGCOMM*, 2014.

[41] George Karakostas. Faster Approximation Schemes for Fractional Multicommodity Flow Problems. *ACM Trans. Algorithms*, 2008.

[42] Robert Krauthgamer, Joseph (Seffi) Naor, and Roy Schwartz. Partitioning Graphs into Balanced Components. In *SODA*, 2009.

[43] Praveen Kumar et al. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *NSDI*, 2018.

[44] Praveen Kumar, Chris Yu, Yang Yuan, Nate Foster, Robert Kleinberg, and Robert Soulé. YATES: Rapid Prototyping for Traffic Engineering Systems. In *SOSR*, 2018.

[45] Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez. Inter-datacenter Bulk Transfers with NetStitcher. In *SIGCOMM*, 2011.

[46] David Lebrun, Mathieu Jadin, François Clad, Clarence Filsfils, and Olivier Bonaventure. Software Resolved Networks: Rethinking Enterprise Networks with IPv6 Segment Routing. In *SOSR*, 2018.

[47] Chansook Lim, S. Bohacek, Joao Hespanha, and Katia Obraczka. Hierarchical Max-Flow Routing. In *Globecom*, 2005.

[48] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *SIGCOMM*, 2014.

[49] Richard McBride. Progress Made in Solving the Multicommodity Flow Problem. *SIAM Journal on Optimization*, 1998.

[50] Aref Meddeb, André Girard, and Catherine Rosenberg. The impact of point-to-multipoint traffic concentration on multirate networks design. *IEEE/ACM Transactions on Networking*, 10(1):115–124, 2002.

[51] Srinivas Narayana, Joe Jiang, Jennifer Rexford, and Mung Chiang. Distributed Wide-Area Traffic Management for Cloud Services. In *SIGMETRICS*, 2012.

[52] NetworkX. Edge Disjoint Paths. https://bit.ly/37VJ71k.

[53] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, 2002.

[54] Murat Oguz, Tolga Bektas, and Julia A. Bennell. Multicommodity flows and Benders decomposition for restricted continuous location problems. *European Journal of Operational Research*, 2017.

[55] James Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Math. Programming*, 1997.

[56] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Victor Bahl, and Ion Stoica. Low Latency Geo-distributed Data Analytics. In *ACM SIGCOMM*, 2015.

[57] H Racke. Optimal Hierarchical Decompositions for Congestion Minimization in Networks. In *STOC*, 2008.

[58] R Tyrrell Rockafellar and Stanislav Uryasev. Conditional Value-at-Risk for General Loss Distributions. *Journal of banking & finance*, 26(7):1443–1471, 2002.

[59] E. Rosen, A. Viswanathan, and R. Callon. Multi-Protocol Label Switching Architecture. RFC 3031.

[60] Matthew Roughan, Albert Greenberg, Charles Kalmanek, Michael Rumsewicz, Jennifer Yates, and Yin Zhang. Experience in Measuring Backbone Traffic Variability: Models, Metrics, Measurements and Meaning. In *IMW*, 2002.

[61] Brandon Schlinker et al. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *SIGCOMM*, 2017.

[62] Submarine Cable Map. http://www.submarinecablemap.com.

[63] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to Route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets, 2017.

[64] Santosh Vempala, Ravi Kannan, and Adrian Vetta. On Clusterings Good, Bad and Spectral. In *FOCS*, 2000.

[65] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. COPE: Traffic Engineering in Dynamic Networks. In *SIGCOMM*, 2006.

[66] I-Lin Wang. Multicommodity network flows: A survey, part i: Applications and formulations. *Internal Journal of Operations Research*, 2018.

[67] Mathieu Xhonneux, Fabien Duchêne, and Olivier Bonaventure. Leveraging eBPF for programmable network functions with IPv6 Segment Routing. In *CoNext*, 2018.

[68] Kok-Kiong Yap et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *SIGCOMM*, 2017.

[69] Jin Yen and NetworkX. K-Shortest Paths. https://bit.ly/2OpNGJn.

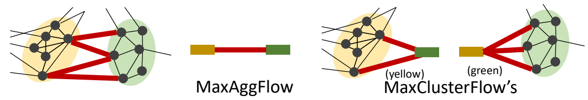


Figure 18: Considering the crossing edges between the yellow and green clusters from Figure 2; MaxAggFlow has a single bundle; the yellow and green instances of MaxClusterFlow have one bundle for each incident node in their cluster.

A Properties of NCFlow's flow allocation algorithm

A.1 Proof that the algorithm in §3.1 meets demand and capacity constraints

Satisfying demand constraints: Demands whose source and target are in the same cluster are considered by only one instance of MaxClusterFlow; hence, they do not receive more flow than their demands. Specifically, MaxClusterFlow in Figure 6 invokes MaxFlow which in turn imposes the demand constraints listed in FeasibleFlow; Equation 1.

Demands whose source and target are in different clusters receive no more flow than their demand due to SrcTargetMax; observe in Figure 6 that one of the four constraints in SrcTargetMax explicitly controls the flow for such demands.

Satisfying edge capacity constraints: We say an edge is local to a cluster if both its incident nodes are within the same cluster. Flow is assigned to a local edge only by the MaxClusterFlow instance of the cluster that contains that edge. Since MaxClusterFlow ultimately invokes FeasibleFlow; by Equation 1 a local edge is allocated no more than its capacity.

Edges that are not local receive flow allocation in MaxAggFlow where, as noted in §3.1, all of the edges that lie between a pair of clusters are treated as a single edge whose capacity equals the sum of the capacity of the underlying edges. Thus, the flow assigned to a bundle of edges by MaxAggFlow is no more than the total capacity of the edges in the bundle. Subsequently, MaxClusterFlow instances behave similarly; that is, the flow allocated for a bundle of edges is no more than the capacity of that bundle. For example, Figure 18 shows the four edges between the yellow and green clusters in Figure 2 as well as the bundles considered by MaxAggFlow (in the middle) and the two instances of MaxClusterFlow corresponding to the yellow and green clusters on the right. The later steps in Figure 6 do no increase flow and so we conclude that capacity constraints are satisfiable for all non-local edges.

A.2 Proof that the heuristic in §3.2 leads to feasible flow allocations

Here, we prove Theorem 1. First, note that the heuristic in §3.2 which only restricts the edges between clusters and paths on the aggregate graph that can be used by some demands does not affect the proof in §A.1; that is, edges still receive flow less than their capacity and demand constraints hold.

We now prove that the heuristic will satisfy flow conservation; that is, at any node in the network, for any demand which neither originates nor ends at this node, the net flow is zero, i.e., incoming flow to the node equals the flow leaving that node.

It is easy to see that flow conservation holds for demands whose source and target are in the same cluster because: (1) Only the instance of MaxClusterFlow for that cluster assigns flow to such a demand. (2) Since MaxClusterFlow invokes FeasibleFlow in Equation 1, the flow is allocated along paths which start and end at the source and target of that demand respectively. (3) Thus, every node that is neither the source or target will have incoming flow equal to the outgoing flow.

We now consider the remaining demands, that is, whose source and target are in different clusters.

It is easy to see that for such demands, flow conservation holds at all nodes that do not have edges to or from other clusters by logic that is similar to the above. The MaxClusterFlow instance of the cluster containing such a node would allocate flow to some bundle of demands on paths in this cluster that neither start nor end at such a node.

The only case left is nodes which have edges to and from other clusters. Suppose by contradiction that some demand k violates flow conservation at such a node u. The heuristic in §3.2 allocates flow for demand k only along one path in the aggregated graph and on only one edge between connected clusters. If the cluster containing u is not on the chosen path or none of the chosen edges are incident on u, then the flow allocated for k on all edges incident on u will be zero. Let e be that one chosen edge incident on u which can receive non-zero flow for demand k. Observe that all of the other demands whose source and target are in the same clusters as k would also be allocated flow on the same path and edges as k. Thus, all the flow allocated for these demands entering or leaving node u as the case may be would be on edge e. Two instances of MaxClusterFlow, one corresponding to the cluster that contains u and another corresponding to the other side of edge e, will assign possibly different flow values for this bundle of demands on edge e. To conclude our proof, note that MinPathE2E takes the minimum flow assigned along all such crossing edges e on the chosen path through the aggregated graph and that SrcTargetMax further breaks open the bundle to assign feasible flow for each actual demand contained in the bundle.

A.3 Proof of optimality for algorithm in §3.1 given some sufficient conditions

Here, we prove Theorem 2. The cases for the number of clusters, η, being 1 or N, the number of nodes in the graph, have already been discussed in §3.3. To prove optimality for the other set of sufficient conditions, we first posit a helper theorem.

Theorem 3. *Given a set of paths $\mathcal{P}$ that can be used by flows, there exists a clustering of nodes into clusters such that any flow allocated by* MaxFlow *on the paths in $\mathcal{P}$ can also be allocated by the method in Figure 6 over those clusters.*

Proof. The claim is trivially true by using N clusters, where each node is in a cluster by itself. We show that it is possible to use fewer clusters next. Let $\mathcal{S}$ be a set of nodes such that every path in $\mathcal{P}$ contains atmost one contiguous sequence of the nodes in $\mathcal{S}$. For example, the set $\{u, v\}$ satisfies this property if every path in $\mathcal{P}$ has neither u nor v, just u but not v (no repetitions allowed), just v but not u, $u \to v$ (no repetitions of u or v anywhere else in the path) or $v \to u$. Coalescing each such set $\mathcal{S}$ into a cluster would allow the method in Figure 6 to allocate the same flow as MaxFlow using the paths in $\mathcal{P}$. □

If G_{agg} is a tree and there is atmost one edge between any pair of clusters, any set of paths $\mathcal{P}$ between node pairs on the actual graph would consist of contiguous segments that are contained within each cluster. Thus, per the above theorem, any flow allocated by MaxFlow on the path set $\mathcal{P}$ can also be allocated by the method in Figure 6.

The only difference then between a global MaxFlow and the method in Figure 6 is that whereas the former is a single optimization call, the latter is a sequence of optimizations.

When demands are satisfiable, every step in the sequence of optimizations of Figure 6 will allocate the maximum amount of flow; that is, they will find routes to carry all of the demands.

B Data-plane details for NCFlow

B.1 Actions at the NCFlow controller, after each allocation

The SDN controller for NCFlow computes total flow per demand and some splitting ratios after each allocation.

Total Flow: The flow assigned to a demand whose source and target are in different clusters is read off SrcTargetMax, i.e., $f_{4,k}$. For intra-cluster demands, their flow is read off MaxClusterFlow, i.e., $f^{x}_{2,k}$ at the cluster x that contains the source and target of demand k. These flow values are summed up over all the iterations used by NCFlow.

Splitting ratios at sources: At source s of cluster x, we have two cases depending on whether the target of the demand is within the cluster x or in some other cluster y.

For the former case, let $\mathcal{P}_{st}$ be the path set to target t for demand k; the splitting ratio for each path p in the set is $f^{x,p}_{2,k}$ summed up over all iterations, divided by the total flow assigned to demand k above. Here, $f^{x,p}_{2,k}$ is the flow assigned to demand k on path p by the MaxClusterFlow instance for cluster x.

For the latter case, let z_i be the next cluster on the one path that can receive flow in iteration i for all traffic going to targets in cluster y. The splitting ratio for path p in the path set $\bigcup_i \mathcal{P}_{sz_i}$ is the value of $\sum_{r \in K_{sy}} f_{2,r}^{x,p}$ summed up over all iterations where K_{sy} is the set of all demands from source s to targets in cluster y divided by the total value for all such paths.

Uniquely, note that each source s has a splitting ratio per target t within the same cluster or per target cluster y.

We call a subset of nodes as ingresses if they have at least one edge to a node in another cluster that is chosen by the offline component of NCFlow in §3.4 as a crossing edge

Splitting ratios at ingresses are computed in a similar way to the splitting ratios at sources. At each ingress node w of cluster y for traffic from cluster x, there are two cases depending on whether the target is some node t in the same cluster as the ingress (y) or in some other cluster z.

For the former case, in iteration i, the splitting ratio for path p in the set $\mathcal{P}_{wt}$ is the value of $\sum_{r \in K_{xt}} f_{2,r}^{y,p}$ in iteration i divided by the total over all such paths. As above, K_{xt} is the set of demands from sources in cluster x to target t.

For the latter case, in iteration i, let z_i be the next cluster on the path to targets in z; the splitting ratio for path p in the set $\mathcal{P}_{wz_i}$ is the value of $\sum_{r \in K_{xz}} f_{2,r}^{y,p}$ divided by the total value over all such paths. As above, K_{xz} is the set of all demands from sources in cluster x to targets in cluster z.

Note that an ingress node w has splitting ratios only for demands whose chosen path at an iteration contains w's cluster (y) and whose chosen edge enters y at w.

B.2 Details on switch forwarding entries

Pathlets: NCFlow sets up label-switched paths (LSPs) between each pair of nodes in each cluster. Which paths to setup is pre-determined by the offline component in §3.4.

Splitting rules: A source s in cluster x has a splitting rule for each other node in the same cluster and for each other cluster. The splitting ratios are as computed in §B.1.

In each iteration, at each cluster, at most one ingress node is active per pair of other clusters. This is because the bundle of demands for a given pair of clusters has at most one crossing edge entering a cluster.

The active ingress node at a cluster x for the bundle of demands from cluster y to cluster z has one splitting rule when $z \neq x$ and one splitting rule per target in cluster x when $z = x$.

Packet content: The LSP (which pathlet to use) is encoded in the L2 header [59]. Additionally, NCFlow has the following tuple in each packet: (x, y, i, e) where x and y are the source and target cluster ids, i is the iteration number of the flow allocation that the packets have been assigned to and e is the edge to leave the current cluster on. The bits needed are $2 \ln \eta + \ln I + \ln \text{node degree}$.[8] We note that 16 bits of header space suffice for all the WAN topologies and experiments considered in this paper; that is $\eta \leq 64$ clusters, $I \leq 8$ iterations and up to 2 edges to nodes in other clusters being used per egress node by NCFlow.

Data path actions:

- At source s in cluster x:
  - The host or middleware adds the cluster-ids x and y into the packet.
  - Source switch uses the appropriate splitting rule to pick a (p, i, e) tuple; the values e and i are placed in the packet and the L2 header gets the identifier for path p.
- Each cluster egress removes e from the packet header and forwards packets to the next-hop of the edge e.
- Each cluster ingress uses the appropriate splitting rule to pick a (p, e) tuple; the value e is put into the packet header and p determines the identifier in the L2 header.

B.3 Definitions of NoMoreFlow

$$\mathsf{NoMoreAlongPath}(\mathbf{f}, \mathbf{f_2}, \mathcal{P}_{\text{agg}}) \triangleq \left\{ \forall k \in \mathcal{D}_{\text{agg}},\ \forall x \in \mathcal{P}_{\text{agg}},\ f_k \leq \sum_{k' \in k} f_{2,k'}^{x} \right\} \tag{3}$$

Here, the flow vectors at a cluster x, $\mathbf{f_2^x}$ allocate flow to different demand bundles than the bundles in $\mathcal{D}_{\text{agg}}$. Here, k' corresponds to the bundled demands that match with the bundled demand k.

$$\begin{aligned} &\mathsf{NoMoreFlowThruCluster}(\mathbf{f}, \mathbf{f_1}, x, \mathcal{P}_{\text{agg}}) \triangleq \\ &\left\{ \forall k \in \mathcal{D}_{\text{agg}},\ \forall y : y \neq x,\ \sum_{p \in \mathcal{P}_{\text{agg}} : (y,x) \in p} f_{1,k}^{p} \geq \sum_{k' \in k,\ p' \in \mathcal{P}_{x,y'*}} f_{k'}^{p'} \right\} \cup \\ &\left\{ \forall k \in \mathcal{D}_{\text{agg}},\ \forall y : y \neq x,\ \sum_{p \in \mathcal{P}_{\text{agg}} : (x,y) \in p} f_{1,k}^{p} \geq \sum_{k' \in k,\ p' \in \mathcal{P}_{x,*y'}} f_{k'}^{p'} \right\} \end{aligned} \tag{4}$$

The first set of constraints ensures that the total flow in $\mathbf{f_1}$ from cluster y to x on any path in $\mathcal{P}_{\text{agg}}$ is no less than the flow that is carried in the MaxClusterFlow instance for cluster x on any of the paths in $\mathcal{P}_x$ that leave the virtual node y' corresponding to cluster y. Note that each bundled demand k from $\mathcal{D}_{\text{agg}}$ on the left can correspond to one or more demands k' on the left.

The next set is identical to the above but for flow from cluster x to cluster y.

[8] The edge id must suffice to distinguish at an egress node between the edges to a particular next cluster; so node degree is an overestimate.

C Benchmarking TEAVAR and TEAVAR*

C.1 Formulation for TEAVAR*

Here, we discuss our adaptation of TEAVAR to maximize total multi-commodity flow. The TEAVAR [19] paper considers a somewhat different objective – maximizing the *concurrent* multi-commodity flow (see Table 2). When all demands are satisfiable, both objectives allocate the same flow; however, when demands require more capacity than is available to meet the desired failure assurance, maximizing total flow leads to a strictly larger allocation. We describe TEAVAR* from first principles here.

In addition to the inputs of MaxFlow (see Equation 2), TEAVAR* has the following inputs:

- A value $\beta \in [0,1]$; larger the value of β, greater the fault assurance.
- A set of fault scenarios, $\mathcal{S}$; each scenario i has a probability of occurrence β_i and a set of failed edges $\mathcal{E}_i$.

In a fault scenario i, the edges in $\mathcal{E}_i$ will fail and so the flow allocated to paths that contain any edge in $\mathcal{E}_i$ will be *lost*. The number of possible fault scenarios is exponential in the number of edges in the network. Thus, to keep the optimization tractable, we consider only a subset of scenarios.

Let $\mathcal{L}(i)$ denote the total flow lost in fault scenario i. Per Proposition 8 in [58], minimizing the potential function, $\alpha + \frac{1}{1-\beta}\mathbb{E}[\mathcal{L}_i - \alpha]^+$, would minimize the conditional value at risk. Here, the expectation is over all possible fault scenarios. Since we only consider a subset of fault scenarios to keep optimization tractable, we minimize: $\alpha + \frac{1}{1-\beta}\left(\sum_{i\in\mathcal{S}}\beta_i[\mathcal{L}_i - \alpha]^+ - (1-\sum_{i\in\mathcal{S}}\beta_i)\alpha\right)$. The last term accounts for the unconsidered scenarios for which we must assume the worst possible loss.

The formulation for TEAVAR* is shown in Equation 5. Recall that f_k^p is the flow assigned to demand k on path p. $\mathsf{Active}_{p,i}$ is an indicator denoting whether path p is active in fault scenario i. Thus, the allocation for demand k in scenario i will be $\sum_{p\in\mathcal{P}_k} f_k^p \mathsf{Active}_{p,i}$. When the allocation is below the required volume d_k, the demand will suffer loss; we use $\mathcal{L}_{i,k}$ to denote the flow loss for demand k in scenario i.

$$
\begin{aligned}
&\mathsf{TEAVAR^*}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P},\beta,\mathcal{S}) && && (5)\\
&\triangleq \arg\min_{\mathbf{f}} \left(\alpha + \tfrac{1}{1-\beta}\left(\textstyle\sum_{i\in\mathcal{S}}\beta_i \mathsf{Excess}_i - (1-\sum_{i\in\mathcal{S}}\beta_i)\alpha\right)\right)\\
&\text{s.t. } \mathbf{f} \in \mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}), && && \text{(Eqn. 1)}\\
&\quad \mathcal{L}_{i,k} \geq 0, && \forall i,k && \text{(loss is non-negative)}\\
&\quad \mathcal{L}_{i,k} \geq d_k - \sum_{p\in\mathcal{P}_k} f_k^p \mathsf{Active}_{p,i}, && \forall i,k && \text{(loss)}\\
&\quad \alpha \geq 0 && && \text{(loss \textit{cutoff})}\\
&\quad \mathsf{Excess}_i \geq 0 && \forall i && \text{(excess loss in scenario } i\text{)}\\
&\quad \mathsf{Excess}_i \geq \sum_{k\in\mathcal{D}} \mathcal{L}_{i,k} - \alpha, && \forall i && \text{(excess loss)}
\end{aligned}
$$

The flow allocation resulting from the above formulation cannot be promised to the demands; in particular, more flow will be assigned on some paths to account for possible failures on other paths. After solving the above LP, we compute the flow allocation for a demand k as follows: (1) sort the per-scenario losses $\mathcal{L}_{i,k}$ in ascending order; (2) starting at index 0, add up the probability of each scenario until the running sum is at least β—let i_β be the unique crossing index; (3) Set demand k's flow to be $d_k - \mathcal{L}_{i_\beta,k}$, the demand minus the loss at the crossing index.

Choosing the fault scenarios to use in TEAVAR*:

- Intuitively, achieving a greater amount of fault assurance requires considering more fault scenarios. Specifically, if the total probability of considered scenarios is below β, the above LP as well as the LP used by TEAVAR become unbounded. To see why, the coefficient of α in Eqn. 5 is $\frac{(\sum_{i\in\mathcal{S}}\beta_i)-\beta}{1-\beta}$. If the probability of considered scenarios is less than β, this coefficient becomes negative, and the objective value reaches $-\infty$ by setting α to ∞.
- Intuitively, if the total probability of considered scenarios is just larger than β, the flow allocated to demands is very small. To see why, the smaller the value of $\sum_{i\in\mathcal{S}}\beta_i - \beta$, the smaller the positive coefficient of α in the objective of Eqn 5. Thus, the solution of Eqn 5 will have a large value of α and a very small amount of allocatable flow.
- In light of these two points, in our experiments, we choose all scenarios that are individually more likely to occur than a cutoff ρ and multiplicatively reduce ρ until the total probability of considered scenarios exceeds $1 - \frac{1-\beta}{2}$.

C.2 Comments on benchmarking TEAVAR

Observe that the number of scenarios affects the complexity of the TEAVAR* optimization; specifically, the number of equations and variables increases by $|\mathcal{S}| * |\mathcal{P}|$. The path set is at least as large as the node pairs, i.e., $|\mathcal{P}| > N^2$ where N is the number of nodes. The appropriate choice of fault scenarios to consider, as discussed above, depends on the size of the topology, the failure probability of edges, and the required assurance level β. Suppose one considers all 2-edge failure scenarios; then $|\mathcal{S}| \sim M^2$ where M is the number of edges. Hence, the increase in equations and variables exceeds N^2M^2. Note that MaxFlow is substantially simpler, having at most $O(N^2)$ variables and constraints (Eqn. 1).

On the topologies listed in Table 5, our implementation of TEAVAR* never ran to completion even after several days. We ran with $\beta = 0.99$ and link failure probability set to 0.004; both of these are the default values used in [3]. The reason is that the optimization problem becomes intractably large. TEAVAR behaves similarly [19]. We conclude that probabilistic fault protection using this methodology is infeasible on large topologies and non-trivial fault assurance levels such as when considering multiple link failures.

We also note that we are unable to simultaneously achieve the solution quality and the runtimes that are reported in TEAVAR [19] using their code [3]. Specifically, achieving the assurance levels reported in their experiments requires many scenarios to be considered. The runtimes reported in [19] appear to have been measured when considering only single link failures.

D Fault Model

When failures happen, several prior works [19, 48] assume that the sources of the label switched paths (LSPs) will proportionally shift traffic. That is, a source that splits traffic in the ratio of $(0.3, 0.5, 0.2)$ between three paths will change to a splitting ratio of $(0.6, 0, 0.4)$ when the middle LSP fails. Doing so can cause congestion on either of the remaining LSPs. The key idea in prior works [19, 48] is to proactively allocate flow such that the maximal load on any link remains under capacity—FFC [48] protects against up to k simultaneous link failures, whereas TEAVAR [19] ensures that the flow at risk is below a given fraction (e.g., 99.9% of flow can be carried by the network on average over all possible failure scenarios).

The cost of such congestion protection is two-fold: (1) proactive schemes substantially increase the solution runtime, and (2) they under-allocate flow, since capacity must be set aside to help with possible failures. Instead, NCFlow uses a *reactive* strategy, and recomputes a new flow allocation after the fault occurs. This enables NCFlow to carry more flow before the fault, and potentially carry more flow after recovery. Furthermore, since NCFlow uses fewer FIB entries for the same number of paths, it is naturally easier to spread flow onto more paths. Thus, the key trade-off is slightly longer and more lossy episodes immediately after a fault when using NCFlow, versus longer runtimes and flow under-allocation with proactive schemes [19, 48].

E Additional Experiments

E.1 Breakdown of NCFlow's Performance

To better understand NCFlow's performance in more detail, we again break down the aggregate results from Figure 11 across various aspects of interest in Figure 19. In the two left-most columns, we break down the results by different settings of α, which illustrates how NCFlow performs on both under-subscribed ($\alpha = \{1, 8\}$) and over-subscribed ($\alpha = \{32, 64, 128\}$) traffic matrices. In the former case, NCFlow is typically able to fully satisfy the TM's requested demand, thereby matching the total flow allocated by the other methods. At the same time, NCFlow is strictly faster on all TMs, except for those belonging to smaller topologies (e.g., Uninett2010), which we discuss later on. As α increases, so, too, does NCFlow's runtime advantage; however, this does come at the cost of the total flow allocated. For example, when $\alpha = 32$, we see many instances where NCFlow is $> 100\times$ faster than PF_4, but allocates 75% of PF_4's total flow in the worst case. This effect becomes more evident for the largest settings of α: here, the speedups are $> 1000\times$, but more flow is sacrificed for some TMs. This behavior occurs because, as the traffic volume increases and the topology becomes more congested, paths that are not allowed by NCFlow's scheme become more critical for maximizing the total flow. In particular, NCFlow does not consider paths that leave and re-enter the same cluster, which leads to more potential bottlenecks within clusters. This reason explains in part a surprising observation about NCFlow's flow allocations: typical bottlenecks occur on edges within clusters, not between clusters.

Topology	Edge-Based	Räcke	KSP	NCFlow
		Total # FIB Entries		
PrivateLarge	945,038,502	52,515,090	22,483,244	**1,694,027**
Kdl	427,524,786	76,794,001	30,199,751	**1,876,289**
PrivateSmall	7,684,182	1,232,866	625,282	**139,346**
Cogentco	7,567,952	2,054,323	915,207	**139,862**
UsCarrier	3,894,542	1,520,821	510,894	**82,301**
Colt	3,534,912	1,048,779	346,905	**67,307**
GtsCe	3,263,696	1,077,350	535,135	**101,368**
TataNld	3,006,720	1,062,629	540,088	**93,179**
DialtelecomCz	2,590,122	1,427,780	529,663	**83,128**
Ion	1,922,000	886,414	418,362	**71,614**
Deltacom	1,417,472	459,159	246,811	**53,948**
Interoute	1,306,910	483,960	249,979	**32,193**
Uninett2010	394,346	133,742	57,428	**21,185**
		Maximum # FIB Entries		
PrivateLarge	962,361	828,397	313,850	**18,124**
Kdl	567,009	576,274	309,575	**16,926**
PrivateSmall	38,809	49,663	21,796	**3,639**
Cogentco	38,416	60,676	30,601	**3,144**
UsCarrier	24,649	41,897	17,822	**2,234**
Colt	23,104	47,077	17,344	**3,572**
GtsCe	21,904	36,070	15,477	**2,748**
TataNld	20,736	24,776	13,179	**2,104**
DialtelecomCz	18,769	34,014	11,084	**1,393**
Ion	15,376	25,261	12,954	**1,387**
Deltacom	12,544	25,135	13,029	**1,737**
Interoute	11,881	14,182	8,346	**710**
Uninett2010	5,329	8,891	3,626	**868**

Table 6: Number of FIB entries for NCFlow vs. edge-based formulations (e.g., Fleischer-Edge), path-based formulations using Räcke Randomized Routing Trees (SMORE*), and path-based formulations using k-shortest paths (PF_4, Fleischer-Path, TEAVAR*) on every topology.

In the middle two columns, we break down the results by traffic model. NCFlow tends to perform best when demands are highly concentrated within clusters. In the bottom middle plot (Poisson, $\delta \to 0$), we see that NCFlow allocates $> 90\%$ of PF_4's total flow for almost every TM, while still achieving speedups $> 100\times$. Recall that, as $\delta \to 0$ in the Poisson traffic model, this decreases the traffic volume *between* clusters, thus generating highly concentrated demands. In contrast, when $\delta \to 1$, demands are less concentrated, which leads to worse performance for NCFlow in terms of total flow, but not in terms of runtime.

Finally, in the two right-most columns, we break down the results by topology size. On Uninett2010, the smallest topol-

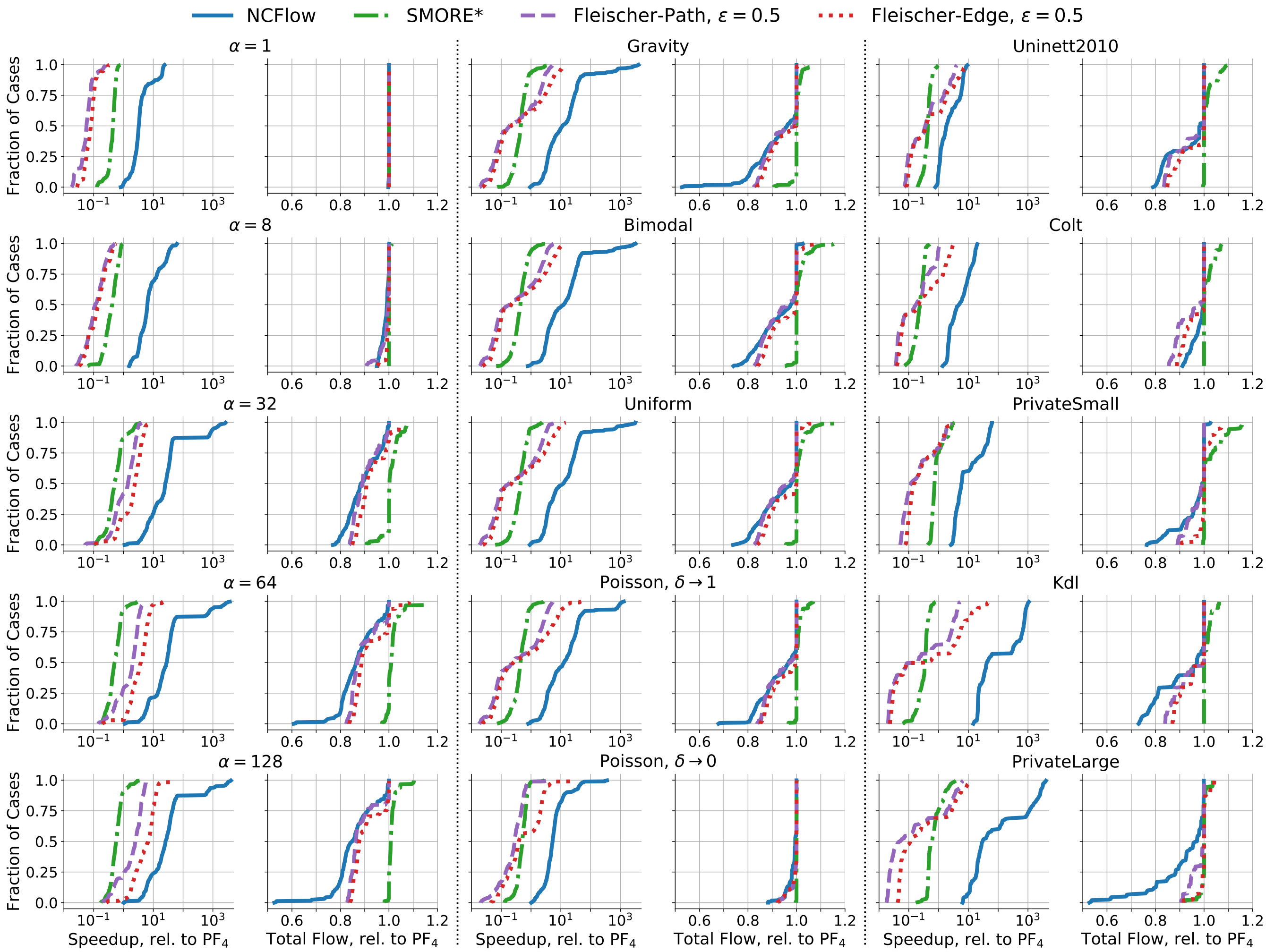


Figure 19: A breakdown of the experimental results from Figure 11 along various dimensions of interest: scale factor, traffic model, and topology size. NCFlow excels on large topologies with TMs that have highly concentrated demands.

ogy in our evaluation set, NCFlow's trade-off between total flow and runtime is no better than the other baselines, particularly Fleischer-Edge. This matches a known limitation of NCFlow, which does not produce significant runtime speedups for small topologies (< 100 nodes).

As the topology size increases, NCFlow's advantage becomes more apparent. On Colt, NCFlow offers faster runtimes and sacrifices little flow, no more than 10% less than PF_4. On PrivateSmall and Kdl, NCFlow's speedup increases even more: $> 100\times$ faster than PF_4 on the majority of cases on Kdl. But more flow is sacrificed, particularly for large values of α. However, NCFlow's trade-off is still favorable compared to other methods: for Kdl, we see multiple instances where NCFlow achieves $1{,}000\times$ speedups at only a 20% reduction in flow. For PrivateLarge, we see both the biggest speedups and the smallest fraction of total flow relative to PF_4. As previously discussed, the outlier coincides with a highly oversubscribed TM ($\alpha = 128$). When we move to other regimes on PrivateLarge, NCFlow's performance improves: on 31 of the 400 TMs with $\alpha \in \{32, 64\}$, NCFlow is $> 1{,}000\times$ faster than PF_4 while achieving > 80% of PF_4's total flow.

In summary, we can see in this panel of CDF plots where NCFlow's strengths lie: on (1) large topologies, and (2) TMs with moderate demand volumes that are highly concentrated within the topology.

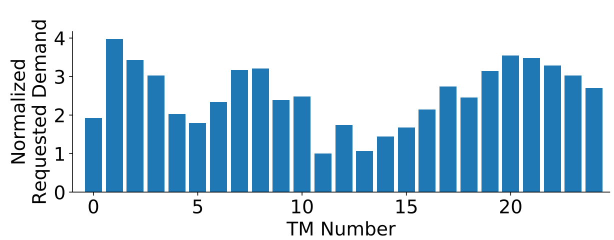


Figure 20: Traffic demand for each traffic matrix used in the demand tracking experiment (see Figure 16) on PrivateLarge. The exact values are not shown for confidentiality reasons.

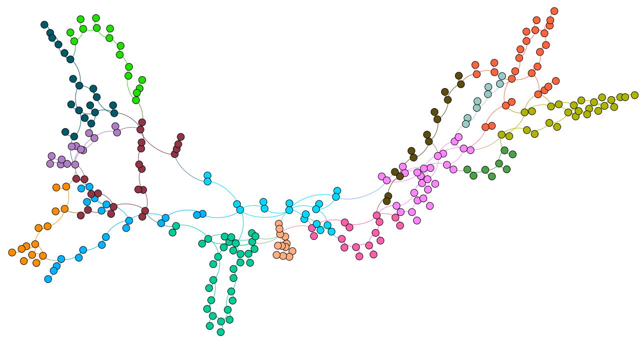


Figure 21: An example clustering, for the GtsCe topology using modularity clustering (described in §3.4).

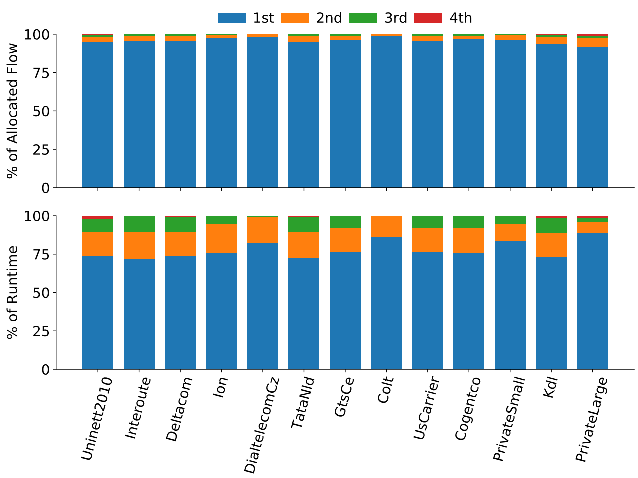


Figure 22: Flow allocated per iteration of NCFlow, which we run with a maximum of 6 iterations.

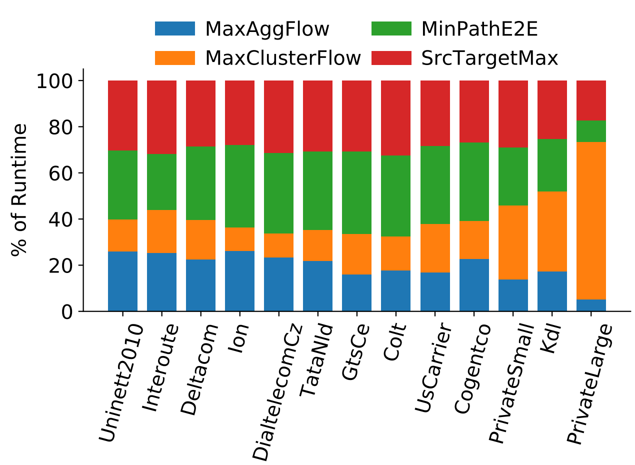


Figure 23: Solver runtime by stage of NCFlow.

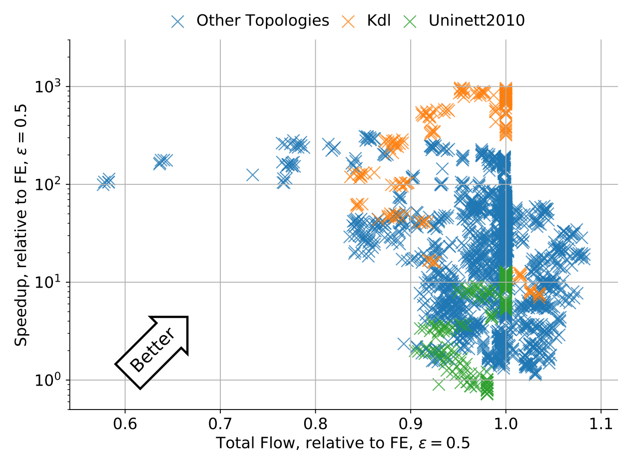


Figure 24: Scatter plot comparing total flow and runtime of NCFlow relative to Fleischer-Edge, $\epsilon = 0.5$. The best- and worst-performing topologies (Kdl and Uninett2010, respectively) are highlighted separately. NCFlow is 28.9× faster and allocates 99.9% of the total flow i the median case.