Beyond Matchings: Dynamic Multi-hop Topology for Demand-Aware Datacenters

Chen Griner[a], Chen Avin[a], Gil Einziger[b]

[a] *School of Electrical and Computer Engineering, Ben Gurion University of the Negev, Israel*
[b] *Department of Computer Science, Ben Gurion University of the Negev, Israel*

Abstract

State-of-the-art topologies for datacenters (DC) and high-performance computing (HPC) networks are demand-oblivious and static. Therefore, such network topologies are optimized for the worst-case traffic scenarios. However, recent optical circuit-switching technologies enable real-time dynamic topologies that change in microseconds or less. This capability opens the door for the design of self-adjusting networks: networks with demand-aware and reconfigurable topologies in which links can be re-adjusted online and in response to evolving traffic patterns.

In this paper, we study self-adjusting networks using a recently proposed model of reconfigurable networks and present a novel algorithm, GreedyEgoTrees (GET), that dynamically changes the network topology. While previous algorithms used a local perspective, GET takes a global view and greedily builds ego-trees for nodes in the network, where nodes cooperate to help each other. In contrast to recent proposals, GET is optimized for multi-hop routing, and we show that it has nice theoretical guarantees as a function of the demand's entropy. Empirical results also show that GET outperforms recently proposed algorithms (like static expander and greedy dynamic matching) and can significantly improve the average path length by up-to 65% for real DC, HPC, and other communication traces.

Keywords: Datacenter Networks, Optical networks, network architecture

1. Introduction

Communication networks, in general, and datacenter (DC) networks, in particular, have become a critical infrastructure in our digital society. The popularity of data-centric applications, e.g., entertainment, science, social networking, and business, is rapidly increasing. The COVID-19 pandemic has further highlighted the need for an efficient communication infrastructure, which is crucial for, e.g., online teaching, virtual conferences, and health [40].

Email addresses: `griner@post.bgu.ac.il` (Chen Griner), `avin@cse.bgu.ac.il` (Chen Avin), `gilein@bgu.ac.il` (Gil Einziger)

Network topology is essential as it is directly related to important network performance metrics such as delay, throughput, and reliability. Consequently, research and innovations in network topologies are fundamental in industry and academia [38, 11, 32, 4, 43]. State-of-the-art (SoA) datacenter network designs typically rely on *static* and *demand-oblivious* optical switches. In particular, static networks often use expander graphs [32] or hierarchical topologies (e.g., FatTree, [4]) to optimize the network connectivity, including the *diameter* and multiple disjoint short paths. These topologies are optimized toward worst-case scenarios like all-to-all communication and assume that the demand can have any pattern. However, DC networks currently serve a *variety* of applications and workloads such as web search, distributed machine learning (ML), High-performance-computing (HPC), or distributed storage. Each application creates different and dynamic demand patterns and the overall traffic is a mix of changing but structured patterns [46, 52, 53, 12, 7].

In recent years, the maturation of optical switching and networks has introduced exciting opportunities for network design. Namely, optical switches that can *dynamically* reconfigure their internal connectivity (input-output port *matching*), which in turn changes the global circuit-switched network topology without physical changes to the network. This new ability has resulted in a flurry of proposals for reconfigurable optical networks [28]. Such networks introduce a *dynamic* topology that optimizes current trends rather than worst-case trends. Thus, dynamic networks can improve network performance if, for example, they create shorter routing paths that lead to higher throughput than static networks in highly structured workloads[39, 27].

There are two main avenues taken by most current proposals. In one type, the network provides direct connectivity between all nodes by rotating between different predefined (i.e. *demand-oblivious*) matchings [38, 11, 37]. In the other type, the network provides a *demand-aware* and *dynamic* topology, based on actual demand [22, 50, 51, 24, 28, 43].

The benefit of the latter approach is that it could be used to take advantage of different localities in network traffic [17]. The dynamic nature of *demand-aware* networks could be leveraged, for example, to reduce the average path length (APL) which measures how many hops flows or packets need to travel in the network. A lower APL enables better link utilization and a higher network *throughput*, as was recently shown [39, 27].

In this work, we adopt the recently proposed ToR-Matching-ToR (TMT) model [27]. This unified model uses a two-layer, leaf-spine, network architecture and, importantly, can describe *both* static, dynamic, demand-oblivious, and demand-aware systems. Figure 1 (left) illustrates the TMT model with seven leaf switches (ToR) and three spine switches (each with a matching) (see Section 3 for formal details).

1.1. **Our Contribution**

Our work demonstrates that demand-aware networks can significantly reduce the *average path length* (APL) on real network traces. While previously proposed algorithms establish only one-hop paths, i.e.,

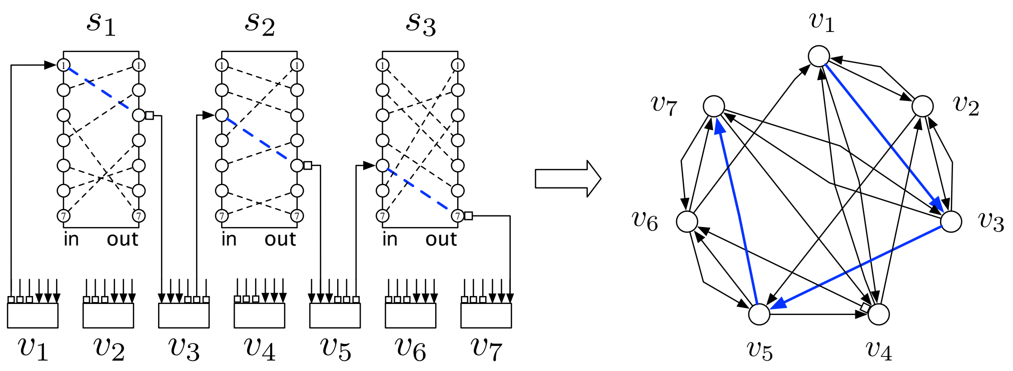


Figure 1: Example of the leaf-spine TMT model (left) with $n = 7$ ToR nodes and $k = 3$ spine switches (matchings) and the corresponding k-regular, directed, network graph at time t, $N(t) = \bigcup_{i=1}^{k} M(i,t)$ (right). The (directed) multi-hop path $v_1, s_1, v_3, s_2, v_5, s_3, v_7$ is shown in the figure. Matchings are reconfigured over time within each spine switch.

direct links, they already improve upon static topologies [13]. Still, we show that using multi-hop paths provides greater benefits. In turn, as our main contribution, we propose a novel offline and online algorithms, GreedyEgoTrees, GET. GET greedily builds parallel ego-trees, trees based on the ego network of nodes, by forming *indirect* links (i.e., multi-hop paths) between sources and destinations. Our network model provides the designer (i.e., our algorithm) a set of k-matchings of size n, one for each spine switch, namely a set of nk direct links that can be reconfigured dynamically. The Algorithm looks at the set of reconfigurable links as a (directed) k-regular graph and not just as a collection of k single-hop matchings. We bound the running time of GET and prove that, for special cases, it has a near-optimal upper bound on the APL, which relates to the *Entropy* of the demand.

We also evaluate Online-GET on ten traces with feasible parameters within the current technological limitations. Our evaluation demonstrates a consistent reduction in APL of up to $\approx 60\%$ compared to static expander networks and up to 30% compared to the state-of-the-art online algorithms. Finally, we note that our work emphasizes a topological perspective of reconfigurable optical networks and abstracts away important networking topics for future studies, such as routing, transport layer and cost analysis.

The rest of the paper is organized as follows: In Section 3, we formally present the network model and the metric of interest. In Section 4 we first discuss static solutions then first present GET. Next, in Section 5 we discuss *online* and *dynamic* algorithms, and extend our algorithms to online setting. Section 6 introduces datasets as well as alternative algorithms . In Section 7, we present our empirical results. After reviewing related work in Section 2 we conclude the paper in Section 8 with a short discussion.

2. Related Work

Today, multi-rooted fat-trees and Clos topologies are some of the most widely deployed static datacenter networks [4, 49]. More recently, Expander graph-based topologies become an important type of static topology which is being evaluated [32]. Expanders are considered state-of-the-art static topologies with better performance (and APL) than, e.g., Clos [4] and Dragonfly [33]. Static topologies are naturally demand-oblivious, but not all demand-oblivious networks are static. For example, RotorNet [38], Opera [37], and Sirius [11] are dynamic optical datacenter network designs that are demand-oblivious. They work by rotating between oblivious and predefined matchings, emulating a complete graph network, and thus providing high throughput. In Opera [37], the authors improved upon RotorNet by ensuring that at every time slot, the network topology is an (oblivious) expander. Opera, therefore combines the benefits of both an expander and RotorNet. However, these designs introduce some difficulties in terms of synchronization and routing. Moreover, they do not change the topology in a demand-aware fashion, and some recent work suggests that demand-aware dynamic topologies may have an advantage over demand-oblivious networks in terms of throughput [27]. In a recent work [43], Google has shown that by using both traffic engineering and what they term as ”topology engineering”, to optimize path length, they can optimize network performance.

A connection between demand-aware network designs and information theory is established in [9, 8]. These studies show that achievable routing performance relies on the (conditional) entropy of the demand. Furthermore, empirical studies point towards traffic patterns in datacenters that are often *skewed* and *sparse* [24, 46]. In addition, an analysis of datacenter traffic in the form of trace complexity shows that network traffic contains patterns that can be leveraged to improve network performance [7]. Meanwhile, several dynamic demand-aware datacenter network designs were proposed, including, Eclipse [50], Mordia [42], or Solstice [36]. These suggestions employ traffic matrix scheduling via Birkhoff-von-Neumann decomposition. The generated schedule of single-hop connection serves all demands in an ideal way, with no limit on reconfiguration. In a recent short paper [35], the authors offer a multi-hop flow *scheduling* algorithm for dynamic networks on a similar model to ours but do not consider the topology design problem. Unfortunately, the paper lacks many details, making any direct comparison to our work impossible. Other projects, such Helios [22], ProjecToR [24], and Online dynamic b-matching [13] and C-Through [51] focus on maximum matching algorithms. None of these designs use indirect multi-hop routing on their dynamic infrastructure.

In particular, b-matching for undirected graphs is quite similar to our approach of online greedy k-matchings, as it optimizes for the highest cost matching. Regarding the offline k-matchings (edge-disjoint) problem (in undirected graphs), in [30], the authors attempt to find an optimal heavy matching, using different offline algorithms, which involve longer running times that depend on k. They also show the k-matchings problem is NP-hard for $k \geq 3$. We note that the b-matching and the k edge-disjoint matchings for undirected graphs are not the same problem. For example, the three edges of a triangle can be covered with a 2-matching, but not with a k edge-disjoint matchings. Another aspect of dynamic networks is what happens when the topology changes. In the paper [54], the authors present a somewhat different approach to evolving topologies on a typical leaf-spine network. They offer TMA, an algorithm for the *management* of both traffic and topology of a dynamic network. In their framework, graphs slowly evolve, and packets can be rerouted through alternate paths. This helps prevent packet loss and helps with the timely delivery of delay-sensitive mice flows. The paper does not study the topology design problem for these networks. CacheNet [26] has recently offered to model demand-aware networks as a network of links-cache and compared this network with demand-oblivious RotorNet [38]. However, while CacheNet is similar to our approach, as we could consider of our set of reconfigurable links as a 'links-cache', it differs in several ways. First, the optimized metric is different. Second, the 'links-cache' in CacheNet is of *unbounded* degree, and third, CacheNet uses only single-hop routing. While there are works exploring a matching-based approach to topology design, e.g., [24, 50, 27], we are unaware of any work exploring an online demand-aware algorithm similar to GET which builds a demand-aware bounded degree *topology* for based on a multihop design.

3. Model and Preliminaries

Our network model is motivated by recent proposals for two-layer leaf-spine network architectures in which spine switches support reconfigurable *matching* between input-output ports [38, 37, 11, 27]. Such architectures are unified under the ToR-Matching-ToR (TMT) model [27] that can model different types of existing systems. For example, Eclipse [50] or ProjecToR [24], which relies on a *demand-aware* switches, RotorNet [38], and Opera [37], which relies on a *demand-oblivious* switches with matchings that rotate over time or an optical variant of Xpander [32] which can be built from a collection of static matchings.

Formally, the network interconnects a set of n nodes $\{v_1, v_2, \dots, v_n\}$ (e.g., leafs, ToR switches) using a set of k optical spine switches $S = \{s_1, s_2, \dots, s_k\}$. Each spine switch has a set of n input and n output ports, internally connected via a directed (i.e., uni-direction) *matching* from the input to the output port. These matchings can be dynamic and change over time. We denote the matching on a switch i at time t by $M(i, t)$.

Each node (i.e., ToR switch) has k uplinks and k downlinks. Given any (leaf) node v_i, its jth up port is connected to the ith input port of spine switch s_j, and its jth down port is connected to the ith output port of spine switch s_j. These links are static and do not change, but the internal matching inside each switch can change. See Figure 1 (left) for an example of the TMT with seven leaf nodes and three spine switches at a given time t.

At each time t our (abstract) network is a union of k-matchings, $N(t) = \bigcup_{i=1}^{k} M(i, t)$. Notice that when all matchings are disjoint *perfect matchings*, having n directed edges, then the resulting network $N(t)$ is always a k-regular *directed* graph with nk edges. Figure 1 (right) shows an example for $N(t)$ which is a 3-regular, directed graph. The network $N(t)$ supports multi-hop routing during time t where a path of length 2ℓ on the TMT network is always of the form $(v_{i_1}, s_{j_1}, v_{i_2}, s_{j_2}, \cdots, s_{j_\ell}, v_{i_{\ell+1}})$ and is translated into a path of length ℓ on $N(t)$ of the form $(v_{i_1}, v_{i_2}, \cdots, v_{i_{\ell+1}})$. Fig. 1 highlights the path $(v_1, s_1, v_3, s_2, v_5, s_3, v_7)$ of length 3 from the source v_1 to the destination v_7, both on the TMT model (left) and on the network $N(t)$ (right).

The network $N(t)$ serves a workload, that is, network traffic represented as a trace of packets or flowlets [41]. Formally, a trace σ is an ordered sequence of communications requests (e.g., IP packets) $\sigma = ((s_1, d_1), (s_2, d_2), (s_3, d_3), \ldots)$, where s_t, d_t represent the source and destination nodes, respectively, and the request (s_{t-1}, d_{t-1}) occurs before the request (s_t, d_t). When the tth request, (s_t, d_t), from a source s_t to a destination d_t arrives, the cost to serve it is assumed to be proportional to the shortest distance (i.e., the number of hops taken by the packet) between s_t and d_t on the network $N(t)$ which we denote as $\text{dist}_{N(t)}(s_t, d_t)$. Recall that our model enables switches to fully reconfigure their connections as long as they form a set of k-matchings, and by that to change $N(t)$ over time. In our model, switches are restricted to *update* their configuration (matching) at a predefined rate $\frac{1}{R} \leq 1$, namely R is the minimum number of consecutive communication requests that are required between two updates. Therefore, our network is static between configurations but could be completely different after an update period (i.e., usually $N(t) \neq N(t+R)$). The update rate accounts for the delay needed to reconfigure modern switches (see Section 7).

A *self-adjusting network* algorithm $\mathcal{A}$ is an *online* algorithm [5] that selects a set of k-matchings that are used to construct the network at time t, namely, $N(t)$. Such *dynamic* algorithms adjust the topology based on a *history* of past requests [9] and use it as an approximation for the near future. We denote by $\mathcal{A}_R$ an algorithm that can make changes at most *once* per R consecutive requests. A *static* algorithm is an algorithm that sets the network (i.e., k-matchings) only once at the start of a trace. We denote this network as N_0. An *offline static* algorithm is assumed to know the whole trace σ (i.e., the future) when it decides N_0.

average path length is defined as the cost to serve an entire trace σ of length $m = |\sigma|$ with respect to an algorithm $\mathcal{A}$ and an update rate R. At each time t the cost of serving a request $\sigma = (s_t, d_t)$ is the shortest path between s_t and d_t in $N(t)$, $\text{dist}_{N(t)}(s_t, d_t)$. Formally, this is the cost metric that we try to optimize,

$$\text{APL}(\mathcal{A}_R, \sigma) = \frac{1}{m} \sum_{t=1}^{m} \text{dist}_{N(t)}(s_t, d_t). \tag{1}$$

In the next section, we discuss two static algorithms, including our proposed novel algorithm GET, and in Section 5 we discuss online algorithms.

4. Static k-Regular DAN

In this section, we explore *static*, *offline*, and *demand-aware* algorithms that yield a static network N for all the traffic. Next, in Section 5, we study the *online* and *dynamic* versions of the problem that construct a dynamic network $N(t)$.

In the static, offline *demand-aware* network design (DAN) problem [8], we receive a demand matrix (DM), $\mathcal{D}$, which describes the distribution of communication requests. Each entry u, v in the matrix is the frequency (or probability), $\text{p}(u, v)$ of a (directed) request from u to v. Alternately, we can assume that the algorithm receives the trace σ as input, and $\mathcal{D}$ describes the empirical distribution of σ. Note that since $\mathcal{D}$ is a distribution, we have $\sum \mathrm{p}(u,v) = 1$. The goal of the offline DAN problem is to design a static network (aka a host graph) $N \in \mathcal{N}_k$, which minimizes the *weighted-average path length* (WAPL). For a given demand matrix $\mathcal{D}$ and a network N, WAPL is defined as:

$$\mathrm{WAPL}(\mathcal{D}, N) = \sum_{(u,v)\in\mathcal{D}} \mathrm{p}(u,v) \cdot \mathrm{dist}_N(u,v). \tag{2}$$

In our model, we require that $\mathcal{N}_k$ is the set of all k-regular directed networks. Formally, the k-regular DAN problem is:

$$\mathrm{DAN}(\mathcal{D}) = \underset{N\in\mathcal{N}_k}{\arg\min}\, \mathrm{WAPL}(\mathcal{D}, N). \tag{3}$$

In contrast to previous work, [30, 50, 24], we aim to design a k-regular network directly. We use the fact that any k-regular directed graph is decomposable to k perfect matchings.

Observation 1. *Any k-regular (multi) directed graph can be decomposed to k perfect matchings (one for each of the k switches).*

This can be proved either by using Hall's theorem [29] or as a special case of the Birkhoff von Neumann Theorem[14]. For the latter, note that the adjacency matrix of a k-regular directed graph of a can be transformed into a doubly stochastic matrix by setting each entry into $\frac{1}{k}$, and the proof follows from the rest of the theorem.

Before presenting our algorithm to build k-regular DAN, we first discuss the *greedy k-matchings* algorithm, which is an appealing but simplistic approach that is the base for the state-of-the-art solutions [30].

4.1. Selfish Approach: Greedy k-Matchings

The weighted k-matchings problem is an extension of the well-known *weighed matching* problem (i.e., $k = 1$) [20, 19]. As is commonly known, a simple *greedy matching* solves the weighted matching problem with an approximation ratio of $\frac{1}{2}$ [10]. It is important to note that the optimization goal of the matching problem (the weight of the matching) is *different* than that of the DAN problem (minimum average path length). Nevertheless, the problems are related since a maximum matching finds a feasible set of requests (that can be served in a single hop) with the maximum probability mass in $\mathcal{D}$. Therefore, the greedy k-matchings algorithm follows the same spirit by greedily building a maximum weight k-regular directed graph using edges with the largest probabilities in $\mathcal{D}$. It starts by sorting requests in $\mathcal{D}$ according to their frequencies (more generally by an ordering $\mathcal{P}$ that can use a different metric). Then, it greedily adds requests according to the order as long as both the source and destination degrees are less than k. Algorithm 1, GreedyMatching (GM), provides pseudo-code for this approach.

Since our optimization problem is different from the weighted matching problem and allows adding edges that are not in $\mathcal{D}$, if we do not yet have a k-regular directed graph at the end of this phase. This is different

Algorithm 1 GM $(\mathcal{D}, k, \mathcal{P})$ Algorithm

```
Require: $\mathcal{D}$ - DM, $k$ - # of switches, $\mathcal{P}$ - ordering
Ensure: $k$-regular DAN
 1: set $N$ = Empty graph                                        ▷ Will be $k$-regular
 2: for each request $(s, d) \in \mathcal{D}$ by the ordering $\mathcal{P}$ do
 3:     if out-deg$(s) < k$ and in-deg$(d) < k$ then
 4:         add $(s, d)$ to $N$
 5:     end if
 6: end for
 7: if $N$ is not $k$-regular and strongly connected then
 8:     add (random) edges to make $N$, $k$-regular, connected
 9: end if
10: convert $G$ to $k$-matchings (each for a switch)
```

from the classical greedy matching that stops and quits. The crucial limitation of the above approach is that it solves the problem from a single, direct-link perspective. This approach leads to a selfish behavior where each node only adds edges for which it is either the *source* or the destination. Therefore, the selfish approach has difficulty handling cases where there is much traffic from a single source to more than k destinations or traffic between more than k sources to a single destination. Such patterns are common in real applications. For example, in the map-reduce framework [16]. Thus, even if many frequently used edges have the same source and varying destinations, Online-GM can only select k edges with the same source.

Fig. 2 demonstrates this problem. Figure 2-(a) presents a weighed demand matrix in a *stars* like structure where both v_1 and v_5 communicate with all other nodes (with different weights). Figure 2 (b) shows the solution imposed by the k-matchings (for $k = 2$). The solution must be a subgraph of $\mathcal{D}$, so only two edges from each star can be included. The GET algorithm we present next overcomes this issue by taking an altruistic approach and adding indirect paths between sources and destinations using *helper* nodes, in particular, other destinations of the same source. Thus, it would still add the most frequent edges in the examples above, but they would not always be direct due to Topological constraints (k regular graph). Figure 2-(c) presents the solution of GET, which we discuss in more detail next.

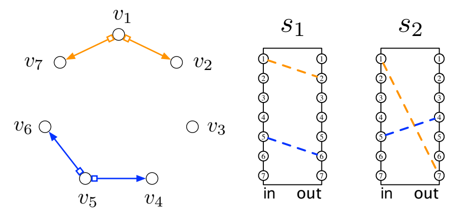
4.2. Altruistic Approach: GET *Network*

We now introduce GreedyEgoTrees (GET), a novel algorithm to solve the offline k-regular DAN problem. While the basic idea of GET follows the spirit of similar algorithms like minimum spanning tree (MST) [44] and greedy matching, GET brings a new networking (or topology) perspective to the proposed solution and has a theoretical foundation that we discuss later.

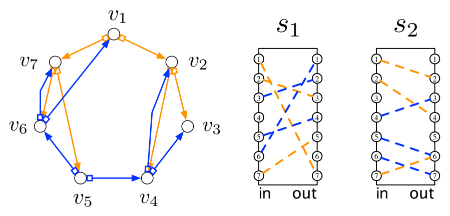
GET *Description.* To build it network $N_{\min}$, GET generates at most $k + 1$ different k-regular networks $\{N_0, N_1, \ldots, N_k\}$. Each network N_i, which we call intermediate, is built from $i \cdot n$ deterministic edges and $(k - i) \cdot n$ random edges. The main ingredient of GET is how it chooses the deterministic edges, which we

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	-	2	1	1	1	1	2
v_2		-					
v_3			-				
v_4				-			
v_5	1	2	2	2	-	1	1
v_6						-	
v_7							-

(a) Demand matrix



(b) The k-matchings solution



(c) GET solution

Figure 2: An example of the difference between k-matchings and GET. (a) a (directed) demand matrix with a star-like structure, only v_1 and v_5 communicate to all other nodes. (b) The result of the greedy 2-matching: the network and the corresponding two matching in s_1 and s_2. (c) The GET solution for the same demand.

explain below. After generating the intermediate networks, GET selects as $N_{\min}$ the intermediate graph with the lowest WAPL. Finally, $N_{\min}$ is converted to k matchings (Theorem 1) defining the concrete switches configurations. See Algorithm 2 for pseudo-code.

More concretely, GET starts by building a network N'. Initially, we sort the requests in the demand matrix, $\mathcal{D}$, according to an ordering $\mathcal{P}$. For now, assume that the ordering is simply the frequency of the requests (we discuss other ordering in Section 7.2). Next, in lines 6 and 5, for each request (i.e., demand from source s to destination d), according to the order $\mathcal{P}$, GET greedily creates *paths* (of length one or more) in the network by adding a *shortcut* that makes s and d closer in the network, if that is possible. We elaborate on this step later. Each time, after adding n new edges to N' (line 11), GET saves the next intermediate graph N_i as the current N', as we see in line 13. GET adds as many paths as possible, it will stop when all requests in $\mathcal{D}$ have a short path in N' or when N' is k-regular (i.e., has kn directed edges, line 10). GET then, in line 17, transforms all intermediate results (i.e., N_i's) into a k-regular directed graphs by adding to each the necessary amount of random edges. We discuss this step in more detail later. In line 20 GET chooses as its final network $N_{\min}$ the intermediate graph with lowest WAPL. In the final step, $N_{\min}$ is converted to k matchings (see Theorem 1) defining the concrete switches configurations.

4.3. Discussion of GET

Let us discuss several of the key ideas behind GET. When GET builds a path for a request (s, d), all previous requests (which had a higher frequency or higher in the ordering $\mathcal{P}$) already have a *short* path between their source and destination. Since the in-degree and out-degree of nodes can be at most k, we may need intermediate nodes to help us add a short path between s and d to N'. We do so by initiating a *forward* Breath-First-Search (BFS) starting at s to find the closest *available* node to s, denoted as x (line 5). That is, we seek the closest node whose *out-degree* is less than k. Note that initially, the closest available node can be s itself. Next, we perform a *backward* BFS starting from d to find the closest available node to the destination d, denoted as y (line 6). That is, a node whose *in-degree* is less than k, and we can therefore add a new edge (x, y) to the network. However, we first verify that adding the edge (x, y) results in a shorter path between s and d than the current network (line 7). Formally, we add the edge (x, y) to N' only if

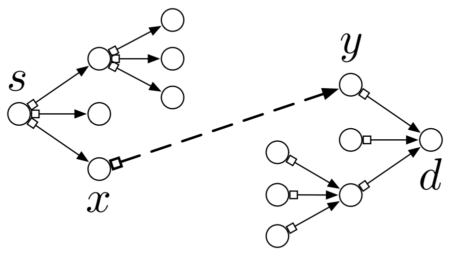


Figure 3: GET Algorithm: Forward BFS and backward BFS example with the corresponding x and y.

$\mathrm{dist}_{N'}(s,x) + \mathrm{dist}_{N'}(y,d) + 1 < \mathrm{dist}_{N'}(s,d)$. See Figure 3 for an example of finding x and y using forward and backward BFSs. If no available nodes x or y exist, we skip to the next request in the ordering.

A second important ingredient of GET is the addition of random edges. A k-regular random graph (i.e., all edges are randomly assigned) is known to be an *expander*, which is a network with many good properties like short diameter and multiple short paths between any two nodes [31]. Therefore adding random edges to our intermediate networks, N_i, can help confer some of these properties to our network.

Recall that N_i, is built from $i \cdot n$ deterministic edges (the first $i \cdot n$ edges added to N') and $(k-i) \cdot n$ random edges (the addition of random edges is done in lines 17-18). Thus, N_0 is k-regular random graph, N_1 has n edges from N' and $(k-1) \cdot n$ random edges, etc. We denoted the last such network, $N_k = N'$, as the *naive* version of GET, and it will not have any random edges (when $|N'| = kn$). As we discuss and demonstrate in Section 7.5, although N_0 is demand-oblivious and naive version $N_k = N'$ (or more generally $N_{i+1} = N'$) is demand-aware (with respect to $\mathcal{D}$), for some demands the WAPL of N_0 can be better (lower) than of N'. In fact, this is the case for any two N_i, N_j for some demand.

To prevent such cases GET finds the lowest WAPL network among all intermediate results between N_0 and N_k. GET returns N, which has the lowest WAPL for the given demand $\mathcal{D}$ among all N_j. This way, GET can never return a worse result than a random expander, which for *any* demand cannot be too bad.

One caveat in our description of GET which we do not present in the pseudo-code is that if a network N is not strongly connected, we identify connected components and connect them by removing low-weight (e.g., probability) edges and adding new edges until we reach a single strongly connected component. For simplicity of presentation, we skip this (solvable) case, and some other minor corner cases (e.g., x or y do not exist) in the pseudo-code description of Algorithm 2.

Fig. 2 (c) presents the result of GET for the demand matrix in 2 (a), ignoring the random edges. We can observe that GET utilizes more edges than the k-matchings approach. Moreover, this simple example builds an optimal directed ego tree, both for v_1 (in orange) and v_5 (in blue).

On a final note, in this work, we focus on the metric of topological shortest paths for network design. Clearly, there are other important network design metrics, like throughput, latency, and maximum congestion. Minimization of the APL leads to optimal throughput and latency only under certain assumptions. In

Algorithm 2 GreedyEgoTrees($\mathcal{D}, k, \mathcal{P}$) Algorithm

```
Require: $\mathcal{D}$ - DM, $k$ - # of switches, $\mathcal{P}$ - ordering
Ensure: $k$-regular DAN
 1: set $N' =$ empty graph
 2: $i = 0$
 3: $N_i = N'$                                                    ▷ $N_0$ is an empty graph
 4: for each request $(s, d) \in \mathcal{D}$ by the ordering $\mathcal{P}$ do
 5:     let $x$ be an available node in ForwardBFS$(N', s)$
 6:     let $y$ be an available node in BackwardBFS$(N', d)$
 7:     if $\text{dist}_{N'}(s, x) + \text{dist}_{N'}(y, d) + 1 < \text{dist}_{N'}(s, d)$ then
 8:         Add $(x, y)$ to $N'$                                  ▷ add a shortcut
 9:     end if
10:     if $|N'| = kn$ then break
11:     if $|N'| \equiv 0 \pmod{n}$ then                          ▷ Every $n$ new edges
12:         $i = \frac{|N'|}{n}$
13:         $N_i = N'$                                            ▷ Save intermediate solution as $N_i$
14:     end if
15: end for
16: $N_{i+1} = N'$                                  ▷ $N$ is the *naive* solution (no random edges)
17: for $j \in [0 \dots i+1]$ do                    ▷ If $nk$ edges were used, $i$ would be $k$
18:     add random edges to $N_j$ to make it $k$-regular
19: end for
20: $N = \arg\min_{N_j : j \in [0 \dots i+1]} \text{WAPL}(\mathcal{D}, N_j)$
21: convert $N$ to $k$-matchings (each for a switch)
```

other cases, using longer paths, which are not congested or even idle, is advantageous over using a shorter congested path. However, recent results show, both for static and dynamic networks, prove a direct connection between the average rout length and throughput [39, 27, 2] which is the main motivation for our work. Furthermore, since random edges are a part of GET, each pair of nodes is likely to have several short and non-shortest paths between them. We therefore believe that the topology created GET should work favorably when used in conjunction with traffic engineering, and not just the shortest path forwarding, to spread the load and increase throughput. We leave this challenging question open for further work.

4.4. Analysis of GET

Next, we discuss several theoretical properties of GET. We start with the following observation to provide insights into the motivation behind GET and the need for less-active nodes or edges to help with high-frequency requests. We denote by $G_{\mathcal{D}}$ the weighted directed graph when we see $\mathcal{D}$ as the adjacency matrix of a directed graph. Consider any demand distribution for which $G_{\mathcal{D}}$ is a *star network* with a root r (i.e., r is a single source, or single destination, for all requests in $\mathcal{D}$). In this case, GET will create N as a k-ary directed (ego) tree network where r is the root and nodes' distance from r is ordered by the frequency they communicate with r. It is easy to see that such a k-ary tree optimally minimizes the weighted-average route length and that $N \in \mathcal{N}_k$. Similarly, we can state the following:

Observation 2. GET *is optimal for a demand $\mathcal{D}$ for which $G_{\mathcal{D}}$ is a collection of disjoint (weighted) stars.*

Next, we extend Observation 2 to the more general demand distribution $\mathcal{D}$ where $G_{\mathcal{D}}$ forms a forest and we bound the WAPL with the *Entropy* [15] of the distribution $\mathcal{D}$.

The information *entropy* (or Shannon entropy) is a measure of the uncertainty in an information source. Since being introduced by Claude Shannon in his seminal 1948 work [48], entropy has found many uses, including coding, compression, and machine learning to name a few [15]. Recently, the conditional entropy was proved to be a lower bound for the APL in static DAN [8]. Formally, for a discrete random variable X with possible values $\{x_1, \dots, x_n\}$, the (base k) entropy $H_k(X)$ of X is defined as $H_k(X) = \sum_{i=1}^{n} p(x_i) \log_k \frac{1}{p(x_i)}$ where $p(x_i)$ is the probability that X takes the value x_i. Note that, $0 \cdot \log_k \frac{1}{0}$ is considered as 0. We can state the following about GET:

Theorem 1. *For $k > 1$ and a distribution $\mathcal{D}$, if $G_{\mathcal{D}}$ is a directed (weighted) forest, then the weighted-average route length of* GET *is less than $H_k(\mathcal{D}) + 1$ where H_k is the entropy (base k) function.*

Proof sketch. First, let $\mathrm{p}_1 \geq p_2 \geq, \cdots \geq p_{|\mathcal{D}|}$ denote the probabilities of the requests in $\mathcal{D}$ in a non-increasing order. Note that it must be the case that $\mathrm{p}_i \leq \frac{1}{i}$, otherwise $\sum_{j=1}^{i} \mathrm{p}_j > 1$, contradiction for $\mathcal{D}$ being a distribution. Next, we show that for the first $|\mathcal{D}|$ edges added to N, N will be a directed forest with max in and out-degree k. Consider the ith request in the sorted list of requests, (s_i, d_i). Since $G_{\mathcal{D}}$ is a forest, the ith request is the only request in $\mathcal{D}$ for which d_i is a *destination*. Thus, the current in-degree of d_i is zero, and when GET finishes, its degree will be one. Since the in-degree of all nodes in N is at most one, and there are no cycles, N will also be a forest. By construction, nodes will have an out-degree of at most k.

Consider what the distance $\mathrm{dist}(s_i, d_i)$ will be after adding the edge (x, y). Following Algorithm 2, $y = d_i$ and x is the closest node to s_i with out-degree less than k. Since N is a directed forest, the sub-tree rooted at s_i can have at most i edges and therefore $\mathrm{dist}_N(s_i, d_i)$ can be at most $\lceil \log_k(i) \rceil$, so $\mathrm{dist}_N(s_i, d_i) < \log_k i + 1$. Overall,

$$\sum_{i=1}^{|\mathcal{D}|} p_i \, \mathrm{dist}_N(s_i, d_i) \leq \sum_{i=1}^{|\mathcal{D}|} p_i \log_k i + 1 \leq \sum_{i=1}^{|\mathcal{D}|} p_i \log_k \frac{1}{\mathrm{p}_i} + 1 \leq H_k(\mathcal{D}) + 1$$

□

Since $\log_k i \leq \log_k \frac{1}{\mathrm{p}_i}$. We note that for the general distribution $\mathcal{D}$, the conditional entropy, $H(X|Y)$ is a lower bound for the APL [8], where X, Y are the sources and destinations nodes, respectively. Such a lower bound can potentially be much lower than the joint entropy $H(\mathcal{D}) = H(X, Y)$ we prove above. Note that after connecting in N all pairs from $\mathcal{D}$, the algorithm will add random edges to create a k-regular directed graph.

We conclude this section by showing that the running time of GET is polynomial.

Theorem 2. *The running time of* GET *is $O(kn^2(k + \log n))$.*

Proof overview. The primary operations in GET are of polynomial time. Sorting can be done in $O(n^2 \log n)$ (or an ordering $\mathcal{P}$ is given). Next, we add edges one at a time to N. Such addition may require the source and destination nodes to construct their BFS tree (forward or backward). BFS runs in $O(m)$ where m is the number of edges. Since we have at most $m = kn$ edges to add, all m BFS searches, two for each edge, can be made at a total time of $O(k^2n^2)$). Additionally, we need to find $\mathrm{dist}_N(s, d)$ in the current network for each path we build. This operation can be done when the source s (or destination d) performs the BFS mentioned above. If there is a path from s to d, it will be found. Adding random edges to each intermediate graph takes no more than $O(kn)$. Calculating a distance matrix for each of the $k+1$ intermediate graphs using Dijkstra should take, along with calculating the WAPL for the demand, no more than $O(n^2(k+\log n))$. This phase is only used for each intermediate graph N_i adding n edges to N we run it k for a run time of $O(kn^2(k+\log n)$. Finally, the graph decomposition (where each maximum matching takes at most $O(nm) = O(kn^2)$) takes no more than $O(k^2n^2)$, meaning the total run time is $O(kn^2(k+\log n))$.

□

In our evaluation in Section 7.7, we will further explore the run time of GET and the other algorithms we evaluate on our dataset. To conclude, we believe that improving the run time of GET is possible, but leave this question for future work. In the next section, we will use our static algorithms as a building block for our discussion on *online* algorithms.

5. Online k-Regular DAN

The online DAN problem denoted as a self-adjusting network [9], deals with cases where the demand matrix or the trace σ is unknown ahead of time. Instead, a past *window* of W requests is used to approximate the current demand matrix.

This section is organized as follows: Section 5.1 presents a *meta*-framework for algorithms for the online DAN problem, Section 5.2 explains how to use GET and GM on top of the framework. Section 5.3 briefly discusses a state-of-the-art algorithm Online-BMA.

5.1. Meta-Algorithm for Online-DAN

All the online demand-aware network (DAN) algorithms we consider in this work follow the same meta-framework to maintain a dynamic network $N(t)$ and to minimize the APL according to Eq. (1). We consider only k-regular DAN so $N(t) \in \mathcal{N}_k$ must be a union of k directed matchings at each time t. Pseudo code for the meta-algorithm is shown in Algorithm 3, and we explain it next. The algorithm receives a trace σ, an update rate R defining the number of requests between subsequent network state updates, and a window size W used to approximate the current demand matrix. In particular, at time t, $\sigma_W = \sigma[t-W, t]$ denotes the window of W last request in σ and $\mathcal{D}_W$ is the empirical demand distribution of σ_W. The update rate R reflects the reconfiguration times imposed by the technological limits of optical switches.

Changing matchings takes time and cannot be executed, for example, after each packet. Therefore, once per R requests, the algorithm updates the network configuration using the *Update*() function (Line 7). The update function computes a new network configuration $N(t+1)$, i.e., a k-regular DAN, based on $\mathcal{D}_W$, the empirical distribution (demand matrix) of the last W requests, k, and an ordering of the request in $\mathcal{D}_W$. The requests can be ordered by frequency or other methods, as we consider in later sections. Next, we consider GET and GM as potential *Update*() functions.

Algorithm 3 Meta-Algorithm for Online-DAN

```
Require: A trace $\sigma$, Update rate $R$, Window size $W$, Ordering function $\mathcal{P}$
Ensure: Dynamic network $N(t) \in N_k$
1: $N(1)$ = Initial network
2: for $t = 1$ to $|\sigma|$ do
3:     Serve $(s_t, d_t)$ on $N(t)$
4:     if $t \equiv 0 \pmod{R}$ then                              ▷ An Update, at rate $R$
5:         $\sigma_W = \sigma[t - W, t]$                              ▷ Set the window
6:         $\mathcal{D}_W$ = empirical distribution of $\sigma_W$
7:         $N(t+1) = \text{Update}(\mathcal{D}_W, k, \mathcal{P}(\sigma_W))$
8:     else
9:         $N(t+1) = N(t)$
10:    end if
11: end for
```

5.2. *Online*-GET *and Online*-GM

The Online-GET algorithm follows the meta-algorithm. Each time the Update method is called, we prepare a new demand matrix D_W based on the requests in $\sigma[t - W, t]$ and an ordering $\mathcal{P}$ and run GET. Formally,

$$N(t+1) = \text{GET}(\mathcal{D}_W, k, \mathcal{P}). \tag{4}$$

In the greedy k-matchings case, we use

$$N(t+1) = \text{GM}(\mathcal{D}_W, k, \mathcal{P}) \tag{5}$$

In the evaluation section, we compare these two methods on real traces. Additionally, we compare both GM and GET with a recent proposal for an online b-matching algorithm [13], which we will describe next.

5.3. *Online b-matching Algorithm*

Online-BMA in an online dynamic version of the classic b-matching problem [6], originally designed for *undirected* graphs. For a directed graph, the problem is identical to the k-matchings problem, where b is the number of switches.

In [13], the authors proposed a state-of-the-art, online competitive algorithm with an approximation ratio of $O(b)$ (in practical settings).

In this paper, we have adopted a directed version of the Online-BMA algorithm [13] to study in this paper. Online-BMA has at its disposal a set of nk reconfigurable links, or edges, which it uses to construct a k-regular network. Whenever a request arrives, if a direct connection exists in the network, it serves it immediately over a single hop. Otherwise, Online-BMA routes the requests using an alternative static expander network.

This means that while the meta-algorithm above uses nk links, Online-BMA uses a topology with a total of at most $2kn$ edges. Another difference from the meta-algorithm is the update rate and the window size. While our algorithms can only update the topology once for every R request, Online-BMA uses a cost parameter α to control the update rate. A cost of 0 means that the network is updated on every request, and higher costs decrease the rate of change. In Section 7, we evaluate Online-BMA with $\alpha = 6$, same as in [13]. This value in practice means that Online-BMA could change edges faster than any of our main algorithms, possibly after only $2\alpha = 12$ request. Online-BMA works greedily by considering a threshold that depends on α. When a source-destination request reaches the threshold, Online-BMA adds that source-destination to the network and, if necessary, removes other edges to keep the degree bounded. We refer the reader to the paper for exact details of the algorithm.

6. Experimental Setup: Datasets and Algorithms

This section introduces the datasets and algorithms that we use in our evaluation. Our evaluation been performed for a topology with $n = 1024$ nodes. The number of switches in the topology, unless otherwise specified, is $k = 3$, which is also the degree of the network. We perform tests with more switches in a later part of the evaluation. Since we are mainly concerned with APL, we do not assume any particular capacity on the links (or edges of the graph topology). However, we do assume that all links are identical. The rest of this section is organized as follows: We begin by discussing the datasets, that is, the traffic traces we used, and then briefly discuss the setting of the tested algorithms in the evaluation section.

6.1. Traffic Traces

We use ten traces from four different sources [1]. Four traces are from a high-performance computing cluster (HPC), three are from a Facebook (FB) datacenter, Two are based on traffic in a university campus [23], and one is a synthetic trace, the star trace, used to present an ideal test case for our algorithm GET. In this trace, the demand comes in the form of several high-degree star graphs. Some of the traces we used are known to contain locality [7], which our algorithms can take advantage of. Table 1 provides some high-level relevant statistics, such as the length of the traces and certain properties of the demand graph of the whole trace. These include the number of unique nodes, the number of directed edges and average, and the minimum and maximum of in- and out-degrees. Later, we will use some of these properties to explain the empirical results. Note that a node that only acts as either a sender or a receiver will have a minimal degree of zero.

HPC traces

Four traces of exascale applications in high-performance computing (HPC) clusters [18]. We refer to these traces as MultiGrid, Nekbone, MOCFE, and CNS, each of these, represents a different application. Figure 4 (a) through (d) provides additional intuition about the traffic patterns of these traces, as they show clear patterns. Some are more ordered than others.

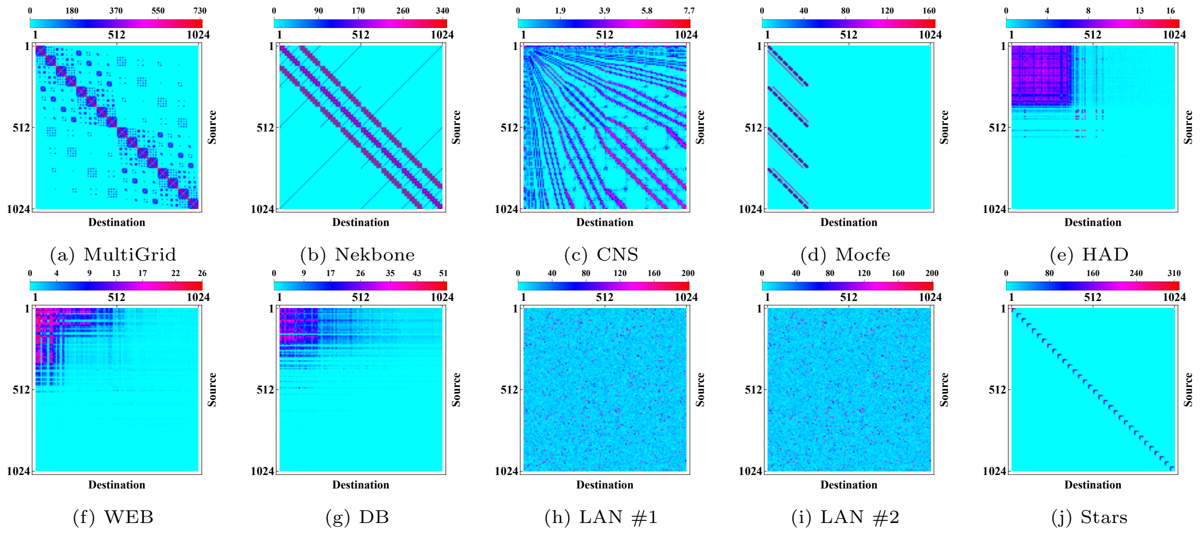


Figure 4: Demand matrices for several of the communication traces. Colors are scaled individually, and the scale is provided at the top of each matrix. Axes represent source IDs (vertical) and destination IDs (horizontal).

Facebook Traces

This set contains three different datacenter clusters from Facebook [46]. The original traces represent more than 300M requests each (including self-loops), with different entries at different aggregation levels, such as pods, racks, and host IPs. The traces we used in this paper are sub-traces that contain 1024 nodes of the most active pairs of the racks level source-destination address trace. The three different clusters represent three different application types, Hadoop (HAD), a Hadoop cluster, web (WEB), a front-end web cluster that serves web traffic, and database (DB), MySQL servers. Figure 4 (e), (f) & (g) show the traffic matrices for these traces. The traces exhibit some structure, but it is less apparent than in the HPC traces.

LAN Traces

Two traces from a university campus LAN network [23]. These traces come from a single 28-minute trace that contains millions of flows. The trace was split into a set of 32 different window traces. Inside each trace, the IP information of the original data was anonymized in a way that removed some spatial locality. This means that each unique request contained a source-destination pair which neither appears in any other unique request. To fit our framework, we transformed the requests so that both the source and destination IDs are in the rage $[1, \ldots, 1024]$. Finally, Since the traces are very similar we present the results of only a couple. These represent two ends of the spectrum in terms of results for our algorithm on this set. Figure 4 (i) & (h) show the traffic matrices for these traces. These traces appear the most random among our trace set, with the matrices being similar to white noise, however, it is also clear that some pairs are much more active than others.

Table 1: Traces

Name	Len	Nodes	Edges	Avg	Min	Max
HPC/MultiGrid	1M	1024	21240	20.74	7	26
HPC/Nekbone	2M	1024	15461	15.1	0	36
HPC/Mocfe	2.7M	1024	4224	4.12	0	20
HPC/CNS	1M	1024	74308	72.56	53	1023
FB/DB	1M	1024	84159	82.18	0	825
FB/WEB	1M	1024	99301	96.97	0	639
FB/HAD	0.8M	1024	154275	150.65	0	577
Synth/Stars	1M	1024	1984	2	1	31
LAN #1	2.6M	1024	100185	97.83	70	131
LAN #2	0.9M	1024	88260	86.19	59	118

Stars Trace

The Star trace is the (only) synthetic trace we use in this work, and we chose it to demonstrate the ideal patterns for GET. Star contains a demand trace from a set of disjoint star graphs of the same size. That is, requests only travel from a star's center to its leaves or vice versa. We used a Zipf-like distribution [45] to determine the traffic distribution in each star. That is, $\frac{C}{i}$ is the probability of a packet to or from the i'th leaf, and C is a normalization constant.

Specifically, our trace uses 32 stars each with 31 leaves resulting in exactly 1024 nodes like the real traces. Figure 4 (j) shows the traffic matrix of this trace.

6.2. Tested Algorithms

Our evaluation includes the three *online* algorithms presented in Section 5, Namely Online-GM, Online-BMA, and our proposed algorithm Online-GET. We also compare the online algorithm with dynamic topologies with two static topologies as a baseline. Specifically, we consider two *static* networks built from k static matching to form a k-regular directed network: i) a *demand-oblivious* expander graph ii) a *demand-aware* offline GET algorithm which assumes knowledge of the entire trace.

Demand-oblivious expander network

Since we are interested in short average path length networks, a natural solution considers networks with short diameters. Expanders [31] that were recently suggested as a datacenter topology [32] are well-known regular graphs with good properties, including large expansion, multiple disjoint paths, small mixing times, and short diameter of $O(\log n)$ where n is the number of vertices. We construct our expander network by creating k, uniformly at random, matchings, one for each switch. It is known from previous work that expanders can be created by taking the union of a few matchings [25].

Demand-aware offline GET

Here we consider the GET algorithm when the whole trace σ is known as an input. Recall that GET creates a demand-aware k-regular network and, in this case, sees the *future* requests. Thus it is an interesting

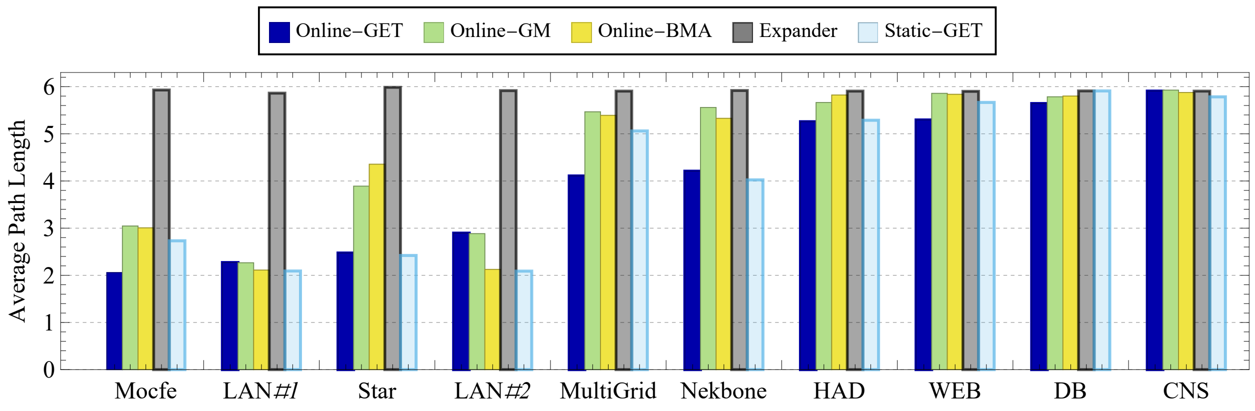


Figure 5: APL results for static algorithms (transparent boxes) and online dynamic algorithms on ten traces. We use $R = 10^4$, and $W = 2\text{x}10^4$ when applicable.

benchmark for Online-GET, indicating how important it is to change configuration dynamically and capture temporal communication patterns within the trace.

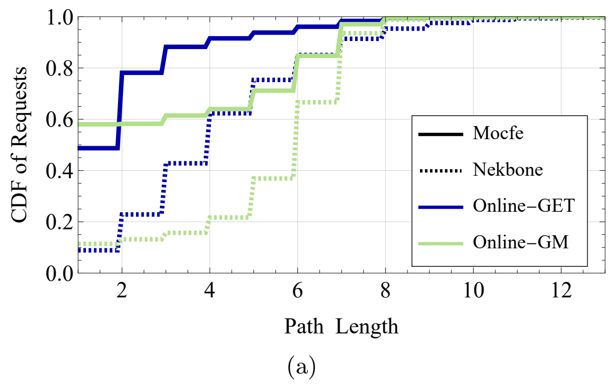
7. Empirical Results

In this section, we evaluate the algorithms on the trace set and explore different configurations. The first section, Section 7.1 shows the main result of our paper, which is the performance of Online-GET in terms of APL compared to other algorithms. Initially we only explore Online-GET with a frequency ordering but in Section 7.2 we also look at other types of orderings. Recall that the number of nodes is $n = 1024$ for all traces, and our default number of switches (matching) was $k = 3$. We discuss other scenarios in Section 7.3. The values of the update rate (R) and window size (W) that were used in the APL evaluation can be found in Section 7.4 where we look at the effect of different values of these parameters on the APL. We discuss how random edges help the GET algorithm in Section 7.5. In Section 7.6 we discuss the number of reconfigurations made by our tested online algorithms to the dynamic topology, and see which algorithm performs more or fewer reconfigurations. Finally in Section 7.7 we experimentally evaluate the running time of the different online algorithms.

7.1. The Average Path Length (APL)

In this section, we discuss our main metric of interest, the APL. We measure APL both for dynamic and static topologies.

The dynamic algorithms use an update rate of $R = 1\text{x}10^4$ and a window of $W = 2\text{x}10^4$ (where relevant). We run the tests on all ten traces described in Table 1 and five algorithms. The three online algorithm for dynamic topologies from Section 5: Online-GM, Online-BMA, Online-GET, and two algorithms for static topologies from Section 6.2 static expander, and Static-GET, where we remind that Static-GET is the offline GET with advance knowledge of the future.



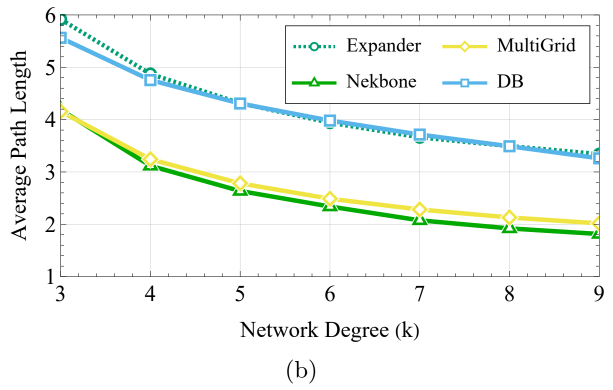


Figure 6: (a) A CDF of path lengths when served by either Online-GET (blue) or Online-GM (green) on two traces Mocfe (continues) and Nekbone (dotted), represented by the solid and dashed lines respectively. (b) APL of Online-GET Vs Expander for an increasing number of switches, k. Note that the APL of the expander is essentially the same for all traces.

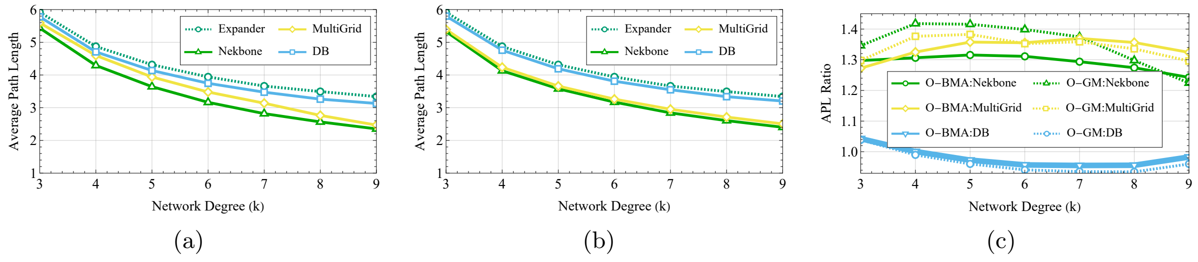


Figure 7: APL of Online-GM (a) and Online-BMA (b) vs Expander for an increasing number of switches, k. The APL of the expander is essentially the same for all traces and is meant as a reference. (c) shows the ratio between the results of Online-GET to Online-BMA and Online-GM. Note that the O in the legend stands for 'Online'

Figure 5 presents the APL for all traces and algorithms (topologies). Full-color bars in the figure mark online dynamic topologies, and transparent bars mark static topologies.

Note, that the bar charts are ordered from the lowest to the highest APL for Online-GET. We note that we don't consider the APL of the first window in these results.

We start with the observation that the expander graph, Unsurprisingly, has an APL of roughly 5.9 for all traces, since the expander is demand oblivious, and on expectation will be the same for all traces. This also shows how demand-oblivious networks can reach consistent results, but fail to take advantage of structured traffic and are optimized for worst-case situations. In contrast, the Static-GET, which is demand aware, is always better than the expander and has up to 65% shorter APL. But, recall that Static-GET has advanced knowledge of the entire trace.

The con of Static-GET is that it can't change its topology, and gain from the temporal locality. Compared to dynamic online algorithms and, in particular to Online-GET.

Figure 5 highlights that dynamic demand-aware networks, such as those represented by Online-GET manage to benefit from temporal-locality, found in these traces, and can beat static algorithms even those which are clairvoyant such as Static-GET. Let us now compare the different online algorithms. In this evaluation, the two matching-based algorithms Online-BMA and Online-GM attain very similar results in all but the synthetic Star trace where Online-GM is better and on the two LAN traces where Online-BMA is better. A possible explanation is that the Star trace has no temporal locality for the algorithms to exploit. On the LAN traces Online-BMA is also better than Online-GM taking advantage of its faster update rate. Figure 5 also reveals that GET has the lowest APL among all the dynamic algorithms with two exceptions on the CNS trace and LAN traces, where the result is in favor of Online-BMA. Recall that Online-BMA uses more edges as it has an additional expander network. In any case, for CNS, GET comes second by less than a 1%. Note also that for CNS there is only a negligible improvement with GET compared to the other algorithms, static or otherwise. For the LAN traces recall also how they are structured. Since, as we discussed before, LAN traces are made from a set of matchings, GET has no advantage when it creates multi-hops paths. Therefore, the benefits of GET could depend on the average degree and the amount of structure found in the trace. These results show that our multi-hop based approach beats the more common k-matching, single-hop approach.

Next, we look into the distribution of path lengths taken by packets on our Online-GET and Online-GM to help us understand how the former beats the latter. Figure 6 (a) shows a CDF of path lengths served by either Online-GET or Online-GM on two traces Mocfe and Nekbone. While Online-GM has more requests sent over a single hop for both traces, a consequence of it optimizing towards this result, Online-GET can optimize towards overall shorter path lengths by sacrificing single-hop connections. Furthermore, on the Mocfe trace, Online-GET can send more than 70% of requests with a path length of 2 or less, showing how this algorithm can take advantage of the structure in the trace and optimize toward lower APL. To conclude, the results show that GET is better and at least no worse than any of the other dynamic algorithms. It most notably outperforms static expander (random graph), representing a current best demand-oblivious topology, by up-to 65%. The results demonstrate locality patterns in real application traces, and that GET manages to leverage these to yield shorter routing paths. Also, observe that GET is better than Online-BMA, which is, in turn, better than demand-oblivious expanders. Our evaluation demonstrates that there are opportunities in demand-aware networks, particularly in dynamic demand-aware networks.

7.2. GET: Pairs Processing Order

An intrinsic part of every demand-aware algorithm is to analyze past demand and use it to predict future demand. So far we have only looked at ordering edges based on the frequency of requests but clearly, there are other possibilities. Another common way to order (or sort) data is by recency. Taking each window of W requests, we order them by their appearance from the end of the window trace. The resulting order is fed into our algorithm at the start of Algorithm 2 as an ordering $\mathcal{P}$. Figure 8 allows us to see a comparison between Online-GET using frequency ordering and recency ordering. We see that recency ordering is comparable to frequency ordering with sometimes even better performance such as on the DB or CNS traces. We note that

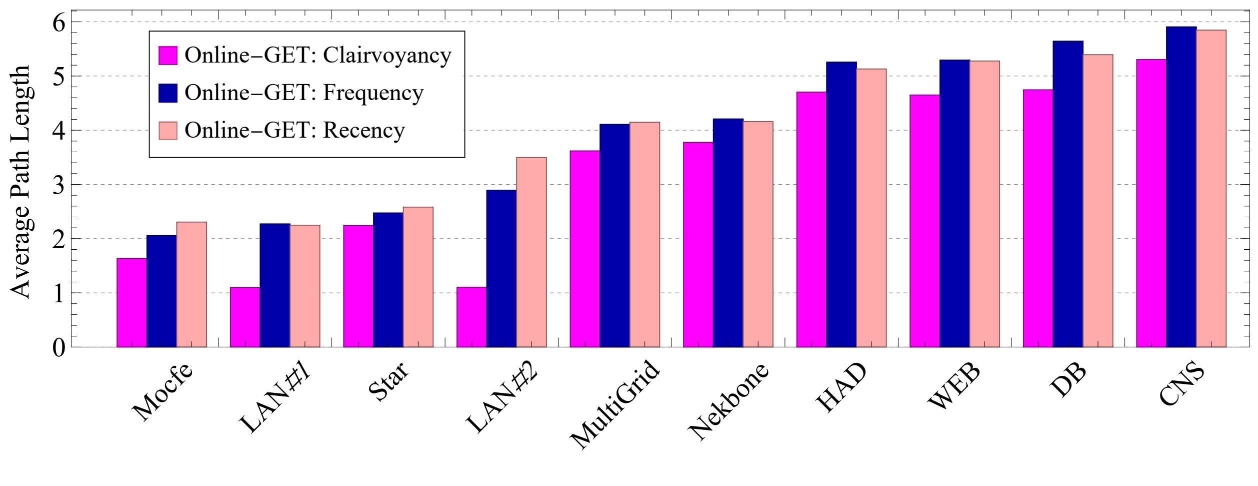


Figure 8: APL results of running Online-GET on all traces with three different edge orderings, by frequency, by recency, and finally by clairvoyancy, where edges are sorted by their order in the next window.

gathering exact statistics of packet frequency could be difficult in a real setting and that recency ordering should require fewer packets to be collected. In Figure 8 we also see the result of an optimal ordering. In this case, requests are ordered according to their *actual* frequency in the next window of requests. The results show that this (impossible) scheme can improve the hop count by close to one hop to almost two hops for the LAN traces, which could be significant [3]. Furthermore, we note here that for the LAN traces, clairvoyant Online-GET beats all other algorithms from Figure 5. While knowing the future is not practical, these results show how better prediction algorithms may achieve better performance with GET. Such prediction algorithms could combine recency and frequency or even use machine learning to predict future traffic, with some algorithms already being developed with similar capabilities [47, 21].

7.3. Higher degree networks, $k > 3$

So far our discussion was limited to a network of degree $k = 3$. In this section, we explore how our online algorithms change in performance in networks with a larger number of switches, that is $k > 3$. We begin with Online-GET and continue with GM and Online-BMA. For each we will compare their APL with our main benchmark, the random expander. Note that when k increases the diameter of the random expander decreases and eventually (for very large k) we expect that all networks will have similar APL. Starting with Online-GET, we see in Figure 6 (b) the APL results of Online-GET on three different traces while increasing the number of switches (matchings) k from 3 to 9. Since the network has more edges, i.e., kn, the window size that GET uses must be increased as well. For a network with k switches we the window size, W_k to be $W_k = (k-1)\text{x}10^4$. From the figure, we observe that the decreasing trend shown by the Expander curve is also closely followed by Online-GET of all traces. The DB trace loses its slight advantage over Expander by $k = 5$, while the gap for both Multigrid and Nekbone from the Expander only slowly decreases. We note that when $k = 9$ Neckbone and Multigrid achieve an APL of around 1.8 which is almost half the APL of the Expander, a better ratio than at the $k = 3$. In Figures 7(a) & (b) we present the same experiment as in Figure 6 (b) for Online-GM (a) and Online-BMA (b) with the alpha parameter set to $\alpha = 6$. For both algorithms, we see that the results are quite similar in nature to the results for GET. We see that the trend is the same, the larger the network degree, the lower the APL. For both algorithms, DB we are slightly better than expander. The fact that this is true for all algorithms and for various k sizes implies that this trace is an example of a trace with a low amount of structure. However, For MultiGird and Neckbone both algorithms improve over Expander but are still worse than for GET. In 7(c) we compare the ratio of the APL of Online-GET to the two other algorithms. We see that while the difference between the APL of both other algorithms becomes smaller, the ratio relative to Online-GET remains similar and nearly constant. For example, on the MultiGrid trace, the APL for $k = 3$ is GET ≈ 4.1, for Online-GM it is ≈ 5.5 and for Online-BMA ≈ 5.2, at least a difference of one hop, and a total of $\approx 26\%$ improvement. for $k = 9$ we have an APL of ≈ 2 for GET, and for both other algorithms, we have an APL of about ≈ 2.5. A difference of half a hop, but this translates to an improvement of 25%. For DB we see that Online-GET is worse than Online-GM or Online-Online-BMA by up to 7% for some value of k. Since Online-GET can install many more edges based on previous history than Online-GM, (since, as you may recall, GET uses trees and is not limited to one hop matchings), it is likely that for this trace, this is counter-productive. We further explore this issue in 7.5. In conclusion, increasing the number of switches in the network lowers the overall APL for algorithms, but Online-GET is able to maintain a relatively stable improvement ratio over the other algorithms.

These results demonstrate that GET is also effective for networks with a larger number of switches, and it is indeed more effective than the other algorithms Online-GM and Online-BMA, even for larger degree networks. We have also seen that even for larger degree networks, GET maintains a similar advantage in terms of the improvement ratio. Recall that in practice we expect that the number of spine switches, k, will be significantly smaller than n, the number of nodes (i.e., ToRs).

7.4. The Window Size and Update Rate

The window size and update rate parameters are what separates dynamic algorithms from static ones in our paper. These dynamic algorithms allow us to take advantage of temporal locality found in network traffic. As we have seen in Section 7.1 a static network that has no updates at all and a future window of the entire trace (i.e., Static-GET), has a worse APL than the dynamic algorithms. Therefore, it is clear that some values for these parameters are better than others. This section will explore the different values of both parameters and how they affect our metric of interest.

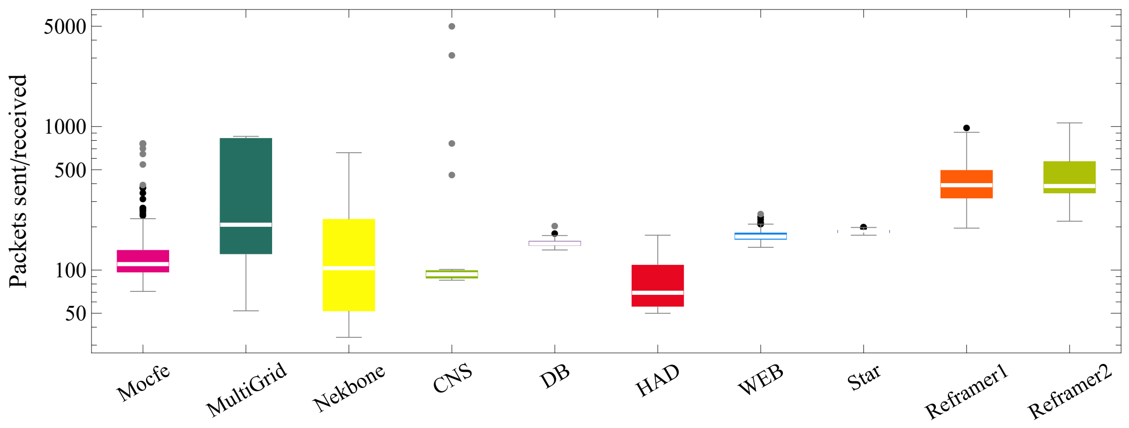


Figure 9: A box plot representing the distribution of most active nodes source/destination inside each update window, using an update rate of $R = 10^4$. Note, that the y axis is log scale.

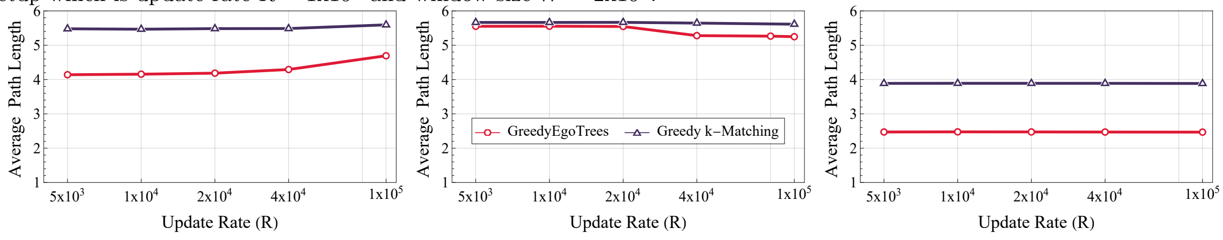
7.4.1. The Update Rate:

The update rate (R) controls the update frequency, and we expect faster updates to yield better results. However, real-world constraints on reconfiguration time would prevent real systems from employing high-speed update rates. Specifically, the reconfiguration time of state-of-the-art optical switches is circa a few tens of microseconds (μs) [28, 42]. Due to a lack of accurate timing regarding packets in our traces, our work measures the update rate in packet counts. We may determine that a realistic reconfiguration time would be anything above once per 10^4 packets according to the following back-of-the-envelope calculation: transmitting a single MTU-sized packet (1500B) on a single $40Gps$ takes roughly $0.3\mu s$. Therefore, we can send approximately 100 such packets on a port within the reconfiguration time of network switches. For 10^4 packets to represent a window of time at least $30\mu s$ long, we would need at least one node to send/receive 100 packets or more during the window. In this case, even if all 100 packets were sent back to back with no delay from the beginning of the reconfiguration window, we would know that enough time has elapsed. Figure 9 explores how common is such an event for each of our 10 traces. The box and whiskers represent the distribution of the number of packets sent/received by the most active node in the window of 10^4 packets, for the whole trace. We can see that for every trace other than for HAD, the average number of packets is at least 100. Furthermore, the distributions show the majority of windows contain at least one high-volume node.

Considering a more empirical approach, in Figure 10 we ran our dynamic algorithms with updated rate values in $\{5\text{x}10^3, 1\text{x}10^4, 2\text{x}10^4, 4\text{x}10^4, 1\text{x}10^5\}$ in the x-axis, and we see how APL responds on the Y axis. In all tests, we used a window size value of $W = 2\text{x}10^4$. Note that lower values than what we think is currently possible were tested in order to investigate if they offer any significant advantage. Figure 10 shows the results for three traces MultiGrid, Hadoop (HAD), and Stars traces. Observe that MultiGrid shows a noticeable increase in APL when increasing the update interval, while the other traces exhibit no change when tested

	Online-GET			**Online-GM**		
Trace Name	Best R	Best W	APL Diff	Best R	Best W	APL Diff
HPC/MultiGrid	0.5x10^4	2x10^4	0.4 %	1x10^4	2x10^4	0 %
Synth/Stars	10x10^4	10x10^4	2.1 %	10x10^4	10x10^4	0.1 %
FB/HAD	10x10^4	4x10^4	6.5 %	10x10^4	2x10^4	2 %

Table 2: The best window size and update rate of different traces and difference of AVL in % from the AVL using the simulation setup which is update rate $R = 1\text{x}10^4$ and window size $W = 2\text{x}10^4$.



(a) MultiGrid (b) HAD (c) Stars

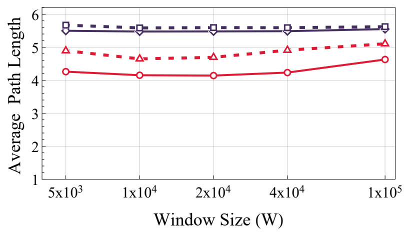
Figure 10: The impact of update rate R. The average path length (APL) of our four dynamic algorithms with increasing update rates on three traces. The same legend is used for all three figures.

with this interval. The MultiGrid workload exhibits temporal locality that we can benefit from with a small enough update rate. On the other hand, the Star trace is completely synthetic and lacks temporal locality as it follows the same communication patterns throughout the trace. looking at the update rate column in Table 2, which summarizes Figure 10 (as well as Figure 11 that we will discuss in the next subsection) we notice two cases. For MultiGird, the optimal update rate is low, 10^4 or 0.5x10^4, while the other two traces, HAD and Stars find an optimal value at 10x10^4. Again, we can attribute this to the existence, or lack thereof of temporal locality in a trace.

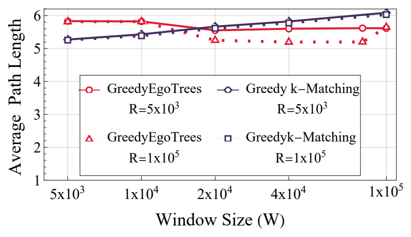
We conclude that while a low update rate is desired, it is not always necessary for a good result. Thus, we performed our evaluation in this paper with an update interval of 10^4 requests which we believe matches current technology capabilities, and, as we can observe in the difference column of Table 2, it is empirically satisfactory in all the workloads yielding at most a small difference in APL.

7.4.2. The Window Size:

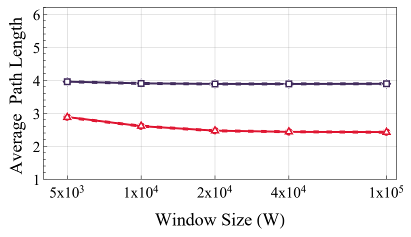
The window size (W) parameter is used in Online-GET and Online-GM. These algorithms use W to estimate the current demand matrix, and it is unclear what is the ideal window size. A short window may be desirable when the demand matrix changes rapidly and significantly over time meaning older demand is no longer an indicator for future demand. When demand is (relatively) static, a larger window yields a better result as it samples the (unchanging) demand more accurately. Figure 11 illustrates how changing the window size (the X axis) for various workloads and at two extremes of the tested update rate range 5x10^3 and 1x10^5 affects the APL (the Y axis). Of the three workloads shown the HAD trace is intriguing, as Online-GM favors a smaller window, while Online-GET favors a large one. Intuitively, Online-GM can only build a



(a) MultiGrid



(b) HAD



(c) Stars

Figure 11: The impact of window size W. The APL of GM and GET for two update rate settings, $R = 5\text{x}10^3$ and $R = 1\text{x}10^5$, and varying window sizes is shown. The same legend is used for all three figures.

smaller number of paths, and thus in small windows, it can exploit temporal-locality. However, Online-GET creates many more (indirect) paths and benefits from a larger window size. Notice that the last data point yields a slightly worse performance implying that Online-GET also has a sweet spot of window size for this trace. Thus, the desired window size depends on the specific matching algorithm used. In Figure 11 (a) MultiGird, we get that Online-GET experiences the sweet spot of Online-GET differently when we change the update rate from 5x10^3 to 1x10^5. This observation implies that the optimal window size also depends on the update rate. Figure 11 (c) Shows how in the synthetic Stars, topology Online-GET continuously improves as we increase the window size, which is predictable since there is no temporal locality in this trace. This is predictable since a small window does not capture all request types. The lack of temporal locality in the Star trace means that there is no benefit in using a small window for this synthetic trace. This is unlikely to be true for most other traces as they usually contain some temporal locality [7].

Table 2 indicated what was the best W and R parameters for three different examples traces. That is, the W, R combination that produced the lowest APL for both of our main algorithms: Online-GET and Online-GM. The last column in the table shows the difference in (%) from the APL produced when using our chosen parameters, which we will discuss shortly. When looking at the window size column at Table 2 we notice that the most optimal values lie at 2x10^4, one exception is the Stars synthetic trace, where the largest tested window size yielded the best result, though by a small margin of 2.1% and 0.1% for Online-GET and Online-GM respectively.

In conclusion, our evaluation demonstrates that the desired window size depends on the workload and the algorithm. However, in the paper, we used a value $W = 2\text{x}10^4$ since, as we can observe in the difference column of Table 2, that value is empirically satisfactory in all the workloads yielding at most a small difference in APL performance.

7.5. Adding Random Edges Can Helps

We now discuss the motivation for adding random edges and the advantage of considering several intermediate networks in GET before selecting the final answer. Recall that in the GET (Algorithm 2, line 20), we compare the intermediate results, N_i,

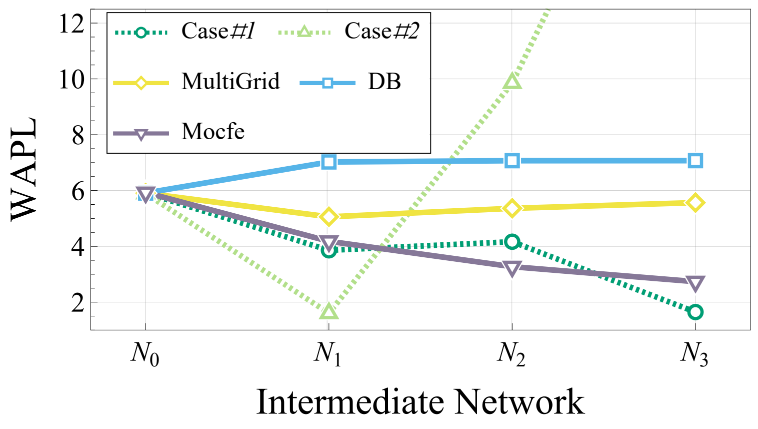


Figure 12: WAPL results of different intermediate networks in Static-GET (Algorithm 2).

and the network which has the best WAPL is chosen. Assume we do not take this step and use only the *naive* version, i.e., the network N_k one, which has the largest amount of deterministic edges, usually kn.

The problem with this network is that we risk 'overfitting' the nk edges in N to the $|\mathcal{D}|$ requests that are in the demand matrix. This could be a bad choice, even when we assume that $\mathcal{D}$ represents perfect knowledge of the future! To see why this might happen, consider the following simplified example, we have two sets of requests, each of size kn requests, one set has a slightly higher frequency than the other in the trace. GET (using frequency ordering) will only add edges for requests from the *first* set to the naive network. In turn, if there are no edges that allow communication between the two sets, the total WAPL will be arbitrarily bad. We give a more detailed and technical example in the Appendix A. For the above particular example, GET will choose N to be N_0, the fully random graph, and avoid the problem.

In Figure 12 we demonstrate such cases. The figure presents the behavior of WAPL (Y-axis) for different intimidated networks in GET (X-axis). Recall that for smaller i, the N_i network contains more random edges. We consider several traces where case #1 and case #2 are syntactic traces for demonstrative purposes (see Appendix A).

For case #1 there is a single minimum at $N = N_1$, that is, with $1/3$ non-random edges. The WAPL is increasing monotonically after this point. We can see similar behavior to this in the real trace, Multigrid. In case #2 we see that there could be a local minimum in the curve. Here the WAPL goes down first at $N = N_1$ giving us a local minimum, then the WAPL goes up slightly and finally finds the global minimum at N_3. For the DB trace, we observe a behavior where the optimal graph is the expander, and GET was not able to improve upon its result. Contrarily to DB, Mocfe shows a case where adding more and more non-random edges strictly improves performance. In conclusion, we see that leaving some room for random edges in our network could be essential.

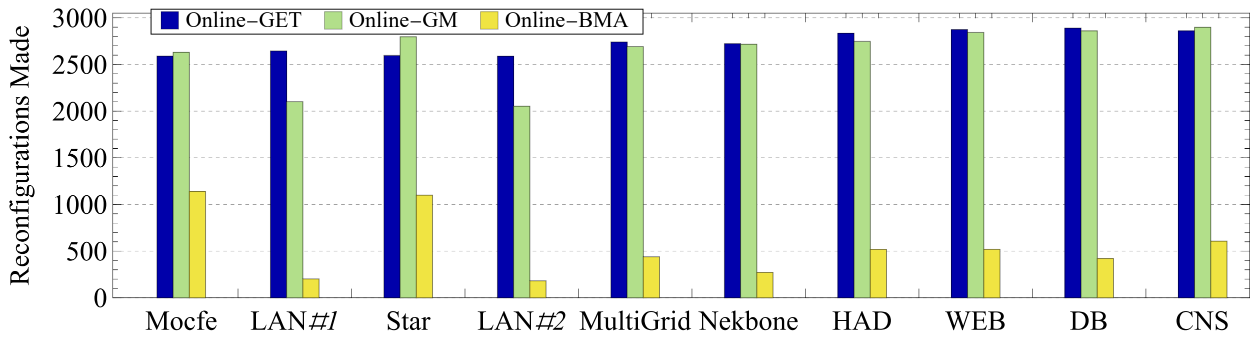
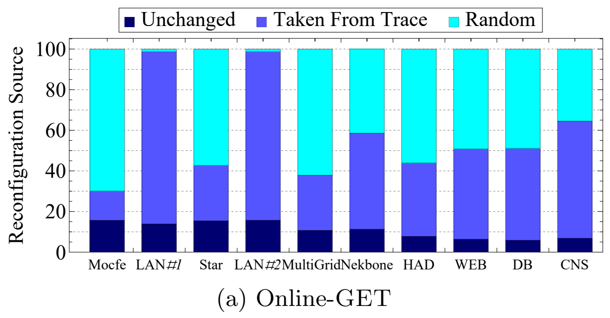
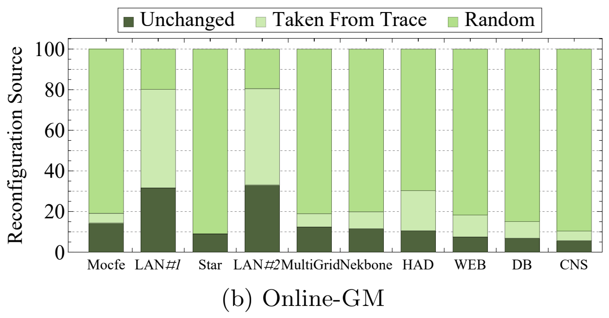


Figure 13: Mean number of edge reconfigurations per 10000 requests, on each of our traces and for our three online algorithms, Online-GET, Online-GM, and Online-BMA.



(a) Online-GET



(b) Online-GM

Figure 14: A subdivided bar chart for each trace. Each bar represents the source of reconfiguration (or lack of it) after each topology update (with $R = 10000$). The division of each bar is based on a percentage of edges that remain unchanged, added from the trace by an algorithm, or added randomly. (a) Online-GET, and (b) Online-GM.

7.6. Comparing Edge Reconfigurations

In this section, we discuss the number of reconfigurations, or topology changes, done by each algorithm on each of our traces and how this number could be minimized. Recall that Online-BMA can potentially reconfigure an edge after every request, while, as we discussed in Section 7.4.1, Online-GET and Online-GM can change their topologies only once after each R request. Reconfigurable network designs often try to minimize the number of reconfigurations, these changes usually draw a cost, temporal or electrical. However, in our work, we assumed that after each update period, the system can reconfigure some or all edges with no extra cost (e.g., the *switch reconfiguration* which changes the matching is the delay cost). Furthermore, recall that Online-GET and Online-GM first add edges based on communication requests appearing in the trace, but after a certain point, switch to adding *random* edges to the topology.

In this evaluation, we measured the number of reconfigurations made for each of our ten traces in total and for each online algorithm. Figure 13 shows the results of these tests. The experiments were done with the same configuration as in Figure 5, that is, with $n = 1024$, $k = 3$, a window size of $W = 20000$ requests, and an update rate of $R = 10000$ requests, for Online-BMA the alpha parameter was set to $\alpha = 6$. The results are normalized towards the update rate. That is, they present the mean number of reconfigurations for each 10000 request from the trace. In Figure 13, We can see that the number of reconfigurations for both Online-GET and Online-GM is approximately $2500 - 2700$ edges (per 10000 requests). This means that, on average, more than two-thirds of the $k \cdot n = 3072$ possible edges are replaced after each update. In Figure 13, we see that Online-BMA makes about 500 to 1000 changes for every 10000 request, less than both other algorithms.

Looking at particular traces, we see that the two LAN traces show fewer reconfigurations, relative to the other traces, for both Online-GM and Online-BMA. To gain a better understanding of the results, we will need to take a closer look at the random edges added to the graphs.

Recall that Online-BMA uses an auxiliary random graph (an expander) to send packets that cannot be sent using the main set of k matchings. It, therefore, avoids adding random edges at any point. However, this is not the case for our other two algorithms. Figure 14 (a) & (b) takes a different look into the results of Figure 13, for Online-GET and Online-GM. In this figure, the colors of each bar represent the source of the edge reconfiguration (or lack of it), by percent, after each topology update by either Online-GET in (a) or Online-GM in (b). More concretely, in each bar, we see the percent of edges (out of the total $nk = 3072$ possible edges in the topology), which are either unchanged, replaced with a random edge, or replaced with an edge from the trace by the algorithm.

Looking just at the results from Figure 14 (a) & (b), we see that Online-GET installs more edges from the trace in its topologies than Online-GM, which uses more random edges. This can be significant, for example, in Neckbone, for and Online-GM, less than 10% of edges are taken from the trace, while for Online-GET, closer to 50% are taken from the trace. In general, most edges, which are replaced, are random edges, and this is more significant for Online-GM, with one main exception: the LAN traces. In Figure 14 (a) & (b), we see that the LAN traces use the least number of random edges, in particular for Online-GET, only around 1% of edges are random. This can be attributed to two aspects of the trace. First, since the traces were anonymized, there is likely no relation between the most popular edges. The second attribute is that there is a very large number of unique requests in the trace, as we show in Table 1, much larger than our update rate. This means that many edges appear only a few times in the trace, and when each of our online algorithms uses them, it will need to replace them at the following topology update, as they likely won't appear again. Unlike the LAN traces, in the Mocfe and star traces, an opposite phenomenon is seen, and Online-GM and Online-BMA have the most reconfigurations

While in this work, we assumed that edge reconfigurations are essentially 'free', we can devise a simple strategy to reduce reconfigurations for both Online-GM and Online-GET. When using random edges instead of sampling new edges, we first sample from constant precalculated set of nk random edges, meaning that the random edges would remain mostly the same after each topology change, which, as we have seen from our results, would reduce the number of reconfigurations. Furthermore, increasing the update window size would also, naturally, decrease the average number of topology changes, particularly when leaving the window size at the same value.

To conclude, we have seen that Online-GET performs a large number of reconfigurations after each update. Taking into account the results of Figure 14 (a) & (b), we have seen that while a significant ratio of these are just random edges, Online-GET can use a much larger fraction of edges taken directly from the traffic trace. Online-GET translates the larger number of reconfigurations to better performance for most traces, as we have seen in Section 7.1. Furthermore, we have described ways to reduce the number of reconfigurations for systems where this is relevant. Since reducing the number of topology reconfigurations was not a focus of this paper, we leave this for future work.

7.7. Algorithms Running Times

This section studies the running time performance of the three online algorithms we evaluated in this work. Recall that we already provided a theoretical analysis of GET in Theorem 4.4. In this section, however, we will compare the actual running time of our online algorithms on our system. Our evaluation was carried out on a computer equipped with an Intel i9-9900 CPU with 32 GB of memory. The code was written in Wolfram Mathematica version 13.2. The running time evaluation was carried out without the benefit of parallel processing, meaning the results are for a single core. Before discussing the results, we note that we did not attempt to optimize the running time of any algorithm, as the focus of the work is the average path length metric. Therefore, these results shouldn't be taken as the best running time of these algorithms on an optimized system. However, we attempt to draw insights by comparing our naive implementations.

Figure 12 shows the running time results for all traces and our three online algorithms: Online-GET, Online-GM, and Online-BMA. The algorithms were tested with the same configuration as in Figure 5, that is, with $n = 1024$, $k = 3$, with a window size of $W = 20000$ requests, and an update rate of $R = 10000$ requests, for Online-BMA the alpha parameter was set to $\alpha = 6$. The Y axis is the mean running time normalized to 10000 requests. Finally, note that the Y axis is log scaled. In the figure, we see that the running time of Online-GM is far lower than the running time of the two other algorithms, where it is approximately 0.2 seconds, while the others are a few tens of seconds, and this is true for all traces. This is because, unlike Online-BMA, Online-GM doesn't need to check for changes for each request, but only once every R requests. Regarding Online-GET, which also only makes changes for every R request, it needs to perform a more complex procedure. In the pseudo-code of GET, in Algorithm 2, we can note that lines 5 thorough 7 require us to perform a BFS twice and then compare the result to an existing distance on the graph. Since the graph is changing, a naive implementation incurs a significant running time. While Online-GM merely looks at vacant ports and doesn't have to make any graph distance calculation. Comparing Online-BMA with Online-GET, we see that Online-GET has a shorter run time on two traces, on the Star

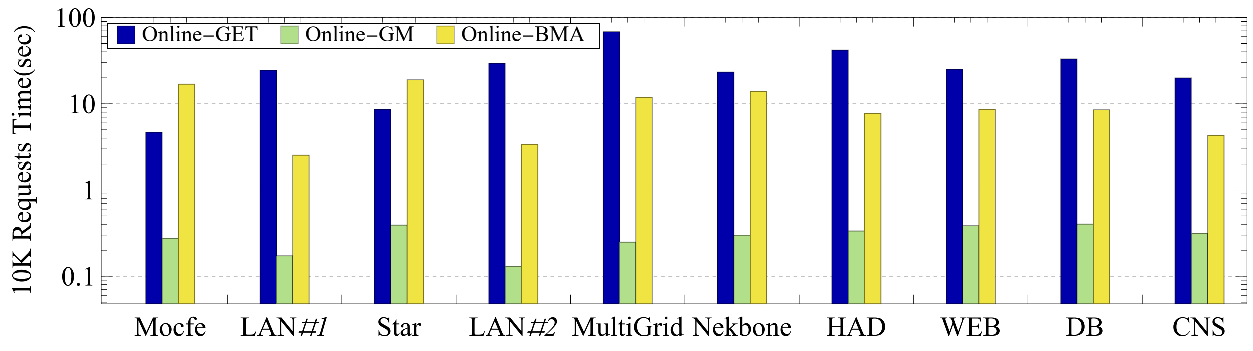


Figure 15: Run-Time results for our three online algorithms. Online-GET, Online-GM, and Online-BMA on all traces. Note the log scale Y axis.

trace and on Mocfe. These two traces also show good APL results with Online-GET when compared to Online-BMA. On these two traces, there are fewer unique requests than for the others, and Online-GET can more easily optimize for the traffic.

We note that a running time of tens of seconds we observe here for 10000 requests is not very realistic either for Online-BMA or for Online-GET. Naturally, each algorithm could benefit from a faster coding language than Mathematica. For Online-GET, in particular, we can offer several optimizations that we leave to future research; GET can be improved by using a more clever bookkeeping scheme to find distances on the slowly updated graph used in the algorithm. Such an improvement might, for example, take inspiration from an algorithm such as D^* Lite [34], which is used, for example, in autonomous vehicles. Another option to reduce run-time is to have an apriori limit on the number of unique requests that we build our network from (more concretely, for line 4 in Algorithm 2, instead of using all requests, we may stop after this apriori limit). Furthermore, we note that the run time is not mainly a function of the length of the trace. Since networks could also benefit from fewer reconfigurations [43], by increasing the update rate size R, we can eventually amortize the run time of the algorithms. In conclusion, Online-GM has a far lower run-time than both other algorithms but worse APL performance. Systems that prioritize running time might benefit from such an algorithm. Furthermore, we see that Online-GET has a run-time on the same order of magnitude as Online-BMA on our framework. Eventually, future work may help reduce the running time complexity of Online-GET.

8. Conclusions

Our work demonstrates that a demand-aware network design can further optimize the network topology and reduce the APL. We present GET that successfully leverages temporal and nontemporal localities in workloads, yielding a shorter APL than static expander-based networks and previous demand-aware algorithms. Specifically, our Online-GET forms short routing paths according to a dynamic demand matrix.

Through extensive evaluations, we show that GET attains up to 65% reduction in APL with respect to the static expander networks. Looking into the future, we seek to form more dynamic algorithms that adapt their configuration most notably, the window size and request weights to the current workload.

We believe that GET can be improved further, for example, adding equal cost paths for some congested links on the basic GET topology may also reduce congestion. However, in general, using alternative routing paths is a difficult problem, which we leave to future research.

Furthermore, it could be interesting to develop a decentralized version of GET where nodes can readjust links based only on partial local information.

Acknowledgement: This project received funding by the European Research Council (ERC), grant agreement no. 864228, Horizon 2020, 2020-2025.

References

[1] Trace collection. https://trace-collection.net/.

[2] Addanki, V., Avin, C., and Schmid, S. Mars: Near-optimal throughput with shallow buffers in reconfigurable datacenter networks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 7*, 1 (2023), 1–43.

[3] Akamai. Akamai Online Retail Performance Report: Milliseconds Are Critical. https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report, 2017.

[4] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM CCR* (2008), vol. 38, ACM, pp. 63–74.

[5] Albers, S. Online algorithms. In *Interactive Computation*. Springer, 2006, pp. 143–164.

[6] Anstee, R. P. A polynomial algorithm for b-matchings: an alternative approach. *Information Processing Letters 24*, 3 (1987), 153–157.

[7] Avin, C., Ghobadi, M., Griner, C., and Schmid, S. On the complexity of traffic traces and implications. *Proc. of the ACM on Measurement and Analysis of Computing Systems 4*, 1 (2020), 1–29.

[8] Avin, C., Mondal, K., and Schmid, S. Demand-aware network designs of bounded degree. *Distributed Computing* (2019), 1–15.

[9] Avin, C., and Schmid, S. Toward demand-aware networking: A theory for self-adjusting networks. *ACM SIGCOMM CCR 48*, 5 (2019), 31–40.

[10] Avis, D. A survey of heuristics for the weighted matching problem. *Networks 13*, 4 (1983), 475–493.

[11] Ballani, H., Costa, P., Behrendt, R., Cletheroe, D., Haller, I., Jozwik, K., Karinou, F., Lange, S., Shi, K., Thomsen, B., et al. Sirius: A flat datacenter network with nanosecond optical switching. In *Proc. of the ACM SIGCOMM Conference* (2020), pp. 782–797.

[12] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *Proc. of the 10th ACM SIGCOMM conference on Internet measurement* (2010), ACM, pp. 267–280.

[13] Bienkowski, M., Fuchssteiner, D., Marcinkowski, J., and Schmid, S. Online dynamic b-matching: With applications to reconfigurable datacenter networks. *ACM SIGMETRICS Performance Evaluation Review 48*, 3 (2021), 99–108.

[14] Caron, R., Li, X., Mikusiński, P., Sherwood, H., and Taylor, M. Nonsquare "doubly stochastic" matrices. *Lecture Notes-Monograph Series* (1996), 65–75.

[15] Cover, T. M., and Thomas, J. A. *Elements of information theory.* John Wiley & Sons, 2012.

[16] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[17] Denning, P. J., and J., P. The locality principle. *Communications of the ACM 48*, 7 (jul 2005), 19.

[18] DOE, U. Characterization of the DOE mini-apps. https://portal.nersc.gov/project/CAL/doe-miniapps.htm, 2016.

[19] Duan, R., and Pettie, S. Approximating maximum weight matching in near-linear time. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science* (2010), IEEE, pp. 673–682.

[20] Edmonds, J. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of research of the National Bureau of Standards B 69*, 125-130 (1965), 55–56.

[21] Eramo, V., Lavacca, F. G., Catena, T., and Salazar, P. J. P. Application of a long short term memory neural predictor with asymmetric loss function for the resource allocation in nfv network architectures. *Computer Networks 193* (2021), 108104.

[22] Farrington, N., Porter, G., Radhakrishnan, S., Bazzaz, H. H., Subramanya, V., Fainman, Y., Papen, G., and Vahdat, A. Helios: a hybrid electrical/optical switch architecture for modular data centers. *ACM SIGCOMM CCR 41*, 4 (2011), 339–350.

[23] Ghasemirahni, H., Barbette, T., Katsikas, G. P., Farshin, A., Roozbeh, A., Girondi, M., Chiesa, M., Maguire Jr, G. Q., and Kostić, D. Packet order matters! improving application

performance by deliberately delaying packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 807–827.

[24] Ghobadi, M., Mahajan, R., Phanishayee, A., Devanur, N., Kulkarni, J., Ranade, G., Blanche, P.-A., Rastegarfar, H., Glick, M., and Kilper, D. Projector: Agile reconfigurable data center interconnect. In *Proc. of the ACM SIGCOMM Conference* (2016), pp. 216–229.

[25] Goldreich, O. Basic facts about expander graphs. In *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation.* Springer, 2011, pp. 451–464.

[26] Griner, C., Avin, C., and Schmid, S. Cachenet: Leveraging the principle of locality in reconfigurable network design. *Computer Networks* (2021), 108648.

[27] Griner, C., Zerwas, J., Blenk, A., Ghobadi, M., Schmid, S., and Avin, C. Cerberus: The power of choices in datacenter topology design-a throughput perspective. *Proc. of the ACM on Measurement and Analysis of Computing Systems 5*, 3 (2021), 1–33.

[28] Hall, M. N., Foerster, K.-T., Schmid, S., and Durairajan, R. A survey of reconfigurable optical networks. *Optical Switching and Networking* (2021), 100621.

[29] Hall, P. On representatives of subsets. *Journal of the London Mathematical Society s1-10*, 1 (1935), 26–30.

[30] Hanauer, K., Henzinger, M., Schmid, S., and Trummer, J. Fast and heavy disjoint weighted matchings for demand-aware datacenter topologies. In *Proc. IEEE Conference on Computer Communications (INFOCOM)* (2022).

[31] Hoory, S., Linial, N., and Wigderson, A. Expander graphs and their applications. *Bulletin of the American Mathematical Society 43*, 4 (2006), 439–561.

[32] Kassing, S., Valadarsky, A., Shahaf, G., Schapira, M., and Singla, A. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proc. of the ACM SIGCOMM Conference* (2017), ACM, pp. 281–294.

[33] Kim, J., Dally, W. J., Scott, S., and Abts, D. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture* (2008), IEEE, pp. 77–88.

[34] Koenig, S., and Likhachev, M. D* lite. In *Eighteenth national conference on Artificial intelligence* (2002), pp. 476–483.

[35] Li, S., Yu, X., Gu, H., and Lu, Y. Nopeak: An intelligent multi-hop scheduling scheme for optical data center. In *2021 19th International Conference on Optical Communications and Networks (ICOCN)* (2021), IEEE, pp. 1–3.

[36] Liu, H., Mukerjee, M. K., Li, C., Feltman, N., Papen, G., Savage, S., Seshan, S., Voelker, G. M., Andersen, D. G., Kaminsky, M., et al. Scheduling techniques for hybrid circuit/packet networks. In *Proc. ACM CoNext* (2015), pp. 1–13.

[37] Mellette, W. M., Das, R., Guo, Y., McGuinness, R., Snoeren, A. C., and Porter, G. Expanding across time to deliver bandwidth efficiency and low latency. In *Proc. of USENIX NSDI* (2020), pp. 1–18.

[38] Mellette, W. M., McGuinness, R., Roy, A., Forencich, A., Papen, G., Snoeren, A. C., and Porter, G. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proc. of the ACM SIGCOMM Conference* (2017), pp. 267–280.

[39] Namyar, P., Supittayapornpong, S., Zhang, M., Yu, M., and Govindan, R. A throughput-centric view of the performance of datacenter topologies. In *Proc. of the ACM SIGCOMM Conference* (2021), pp. 349–369.

[40] OECD. Keeping the internet up and running in times of crisis.

[41] Perry, J., Balakrishnan, H., and Shah, D. Flowtune: Flowlet control for datacenter networks. In *Proc. of USENIX NSDI* (2017), pp. 421–435.

[42] Porter, G., Strong, R., Farrington, N., Forencich, A., Chen-Sun, P., Rosing, T., Fainman, Y., Papen, G., and Vahdat, A. Integrating microsecond circuit switching into the data center. In *Proc. of the ACM SIGCOMM Conference* (2013), pp. 447–458.

[43] Poutievski, L., Mashayekhi, O., Ong, J., Singh, A., Tariq, M., Wang, R., Zhang, J., Beauregard, V., Conner, P., Gribble, S., et al. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 66–85.

[44] Prim, R. C. Shortest connection networks and some generalizations. *The Bell System Technical Journal 36*, 6 (1957), 1389–1401.

[45] Reed, W. J. The pareto, zipf and other power laws. *Economics letters 74*, 1 (2001), 15–19.

[46] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. In *Proc. ACM SIGCOMM CCR* (2015), vol. 45, ACM, pp. 123–137.

[47] Salman, S., Streiffer, C., Chen, H., Benson, T., and Kadav, A. Deepconf: Automating data center network topologies management with machine learning. In *Proceedings of the 2018 Workshop on Network Meets AI & ML* (2018), pp. 8–14.

[48] Shannon, C. E. A mathematical theory of communication. *Bell system technical journal 27*, 3 (1948), 379–423.

[49] Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM CCR 45*, 4 (2015), 183–197.

[50] Venkatakrishnan, S. B., Alizadeh, M., and Viswanath, P. Costly circuits, submodular schedules and approximate carathéodory theorems. *Queueing Systems 88*, 3-4 (2018), 311–347.

[51] Wang, G., Andersen, D. G., Kaminsky, M., Papagiannaki, K., Ng, T., Kozuch, M., and Ryan, M. c-through: Part-time optics in data centers. *ACM SIGCOMM CCR 41*, 4 (2011), 327–338.

[52] Wang, W., Khazraee, M., Zhong, Z., Jia, Z., Mudigere, D., Zhang, Y., Kewitsch, A., and Ghobadi, M. Topoopt: Optimizing the network topology for distributed dnn training. *arXiv preprint arXiv:2202.00433* (2022).

[53] Zhang, M., Zhang, J., Wang, R., Govindan, R., Mogul, J. C., and Vahdat, A. Gemini: Practical reconfigurable datacenter networks with topology and traffic engineering. *arXiv preprint arXiv:2110.08374* (2021).

[54] Zhao, Y., Wang, S., Luo, S., Anand, V., Yu, H., Wang, X., Xu, S., and Zhang, X. Dynamic topology management in optical data center networks. *Journal of Lightwave Technology 33*, 19 (2015), 4050–4062.

Appendices

Appendix A. Synthetic Traces to Understand Why Random Edges Helps

This section expands on the edge cases discussed in Section 7.5. Consider the following example for $k = 3$. We have two sets of three request types, note that each type here is a matching. The first is of the form $\{i, (i+1) \bmod n\}, \{i, (i+2) \bmod n\}, \{i, (i+3) \bmod n\}$, that is each node $i \in [0, ..., n-1]$ communicates with three of their closest neighbors in the order of IDs, and the last set is $\{i, (i+n-3) \bmod n\}, \{i, (i+n-2) \bmod n], \{i, (i+n-1) \bmod n\}$. That is, each node communicates with its three furthest neighbors. Each of the two sets contains $kn = 3n$ edges, therefore able to occupy the entire network. Assume we form a trace with $3l + 3$ requests from the first set ($l + 1$ for each type) and $3l$ requests from the second set (l for each type). If GET orders requests by frequency, then only the edges from the first set are added to the graph. When serving the trace, requests from the first set are served within one hop, while requests from the second set are served at $\frac{n}{3}$ hops. The WAPL for this trace will be $\frac{1\cdot(l+1)+\frac{l\cdot n}{3}}{2l+1} = O(n)$. With generalization to k switches, this formula will equate to $o(\frac{n}{k})$, which could be problematic since for expander, the worst case limit is $o(d)$ where d is the graph diameter, which is generally $\log n$. Let us also describe the traces for case #1 and #2 from Figure 12.

Case #1:

The matching edge sets are of the form:

- $b_1 = \{(i, (i+1) \bmod n)\} \forall i \in [0, ..., n-1]$.

- $b_2 = \{(i, (i+2) \bmod n)\} \forall i \in [0, ..., n-1]$.
- $b_3 = \{(i, (i-2+\frac{n}{2}) \bmod n)\} \forall i \in [0, ..., n-1]$.
- $b_4 = \{(i, (i-1+\frac{n}{2}) \bmod n)\} \forall i \in [0, ..., n-1]$.
- $b_5 = \{(i, (i-1+n) \bmod n)\} \forall i \in [0, ..., n-1]$.

This trace is constructed with the following frequencies. Staring with b_1, its frequency is $|b_1| = l$ that is, there are l edges from this set. For the other edges, we need the following frequencies (relative to l). $|b_2| = l-1$,$|b_3| = l-2$,$|b_4| = l-3$,$|b_5| = l-4$.

Case #2:

- $b_1 = \{(i, (i+1) \bmod n)\} \forall i \in [0, ..., n-1]$.
- $b_2 = \{(i, (i+2) \bmod n)\} \forall i \in [0, ..., n-1]$.
- $b_3 = \{(i, (i+3) \bmod n)\} \forall i \in [0, ..., n-1]$.
- $b_4 = \{(i, (i-2+\frac{n}{2}) \bmod n)\} \forall i \in [0, ..., n-1]$.
- $b_5 = \{(i, (i-1+\frac{n}{2}) \bmod n)\} \forall i \in [0, ..., n-1]$.
- $b_5 = \{(i, (i-1+n) \bmod n)\} \forall i \in [0, ..., n-1]$.

This trace is constructed with the following frequencies. Staring with b_1, its frequency is $|b_1| = l$. That is, there are l edges from this set. For the other edges, we need the following frequencies (relative to l). $|b_2| = \frac{l}{2}$,$|b_3| = \frac{l}{2}$,$|b_4| = \frac{l}{2}-1$,$|b_5| = \frac{l}{2}-1$,$|b_6| = \frac{l}{2}-1$