Reproducing Scafida: A Scale-free Network Inspired Datacenter Topology

Stylianos Rousoglou
steliosr@stanford.edu

1 INTRODUCTION

Large datacenters have been at the heart of the increasingly content-centric internet for the past decade. Many architectures have been proposed, such as fat-tree, BCube, and Jellyfish, a choice which does not come without tradeoffs. On the one hand, fat-trees are deterministic, symmetric architectures with constant inter-server path lengths and minimal room for implementation errors. Symmetric architectures work well for pre-planned, fixed-size data centers when the equipment is homogeneous and known in advance. However, these assumptions break down once we factor in incremental expansion as a requirement. Jellyfish attempts to mitigate this problem by following a quasi-random iterative approach to incrementally add equipment to the network. However, after building the initial network, and given that edges are not removed once added (with a very specific exception), it's still unclear whether the process of incrementally expanding Jellyfish maintains the desired properties of random graphs at scale, such as short paths, or too heavily depends on the existing network. Moreover, Jellyfish doesn't address the problem of homogeneous equipment: all switches are assumed to have the same number of ports, and servers only have 1 port, sitting at the edge of the network as a result.

In this paper, we reproduce the main findings for Scafida, an alternative datacenter architecture that breaks the symmetry traditionally found in topologies, while allowing for arbitrary numbers of heterogenous equipment in the network and involving servers with more than one port in the routing process. The Scafida topology is inspired by scale-free networks (e.g. ordinary Barabasi-Albert) and all the desirable properties that come along, namely short paths and high fault tolerance (path redundancy.) However, it imposes the additional constraint of limiting the degrees of nodes in the network, since neither switches nor servers can have arbitrary numbers of ports. The original paper finds that such degree constraints have little negative effect on the resulting topology, and find that Scafida has comparably favorable properties, such as mean path lengths, diameter, and bisection bandwidth, to the state-of-the-art topologies, despite its asymmetric structure.

Our reproduction starts with understanding the ordinary Barabasi-Albert scale-free network, and implementing the Scafida variation to limit node degrees and parametrize for heterogeneous equipment; as in the Scafida paper, nodes are added in the network prior to being designated as switches or servers. The iterative process with some backtracking is inherently slow, and the significant number of data structures involved in tracking the growing network increases the complexity and runtime of the implementation, written in Python 3. Standard algorithms are ran on the generated topology to compute mean path lengths, bisection bandwidth, and amount of disjoint paths, and the process repeats so that the results presented are averaged over all simulations. Finally, other datacenter architectures are produced, and maintaining the same amount of servers, their metrics are compared to Scafida's. As of this intermediate report, the algorithm implementation and preliminary node degree results that match the original paper very closely suggest that the implementation is likely correct.

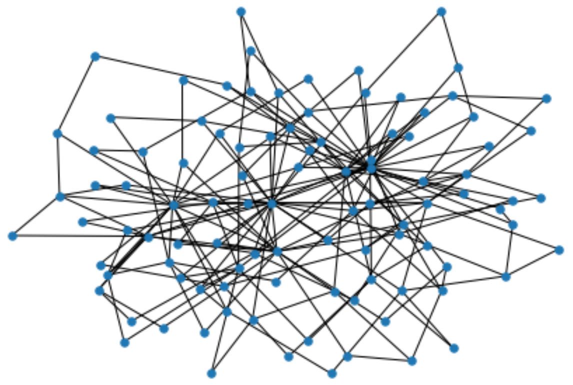
2 REPRODUCTION

Figure 1a shows a Barabasi-Albert scale-free network of 100 nodes without degree limitation; figure 1b shows plots a Scafida topology with the same number of nodes, with 60 servers; 10 switches of degree 4; 10 switches of degree 8; and 20 switches of degree 16.

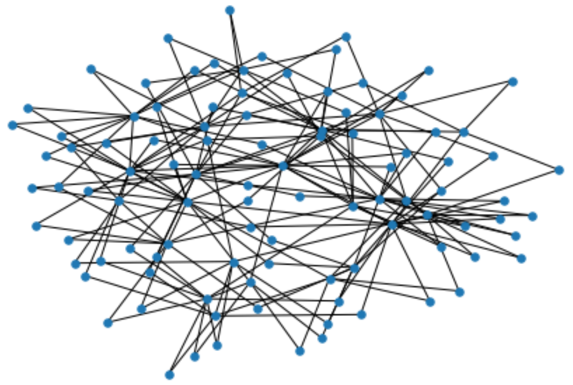
Figure 2 shows a reproduction of figure 4a, which plots the average shortest path length for Scafida topologies with different amounts of nodes and a different number of ports on the commodity switches (8, 16, 24, 48, No Limit), computer by diving the sum of the shortest path lengths by the number of shortest paths. The presented values are averaged over 50 topologies generated with the Scafida algorithm, and the value of m (degree of servers) is set to 2, which implies servers may be involved in the routing process. The original figure is presented along with the reproduced figure; it's apparent that they resemble each other quite closely, which suggests the Scafida (re-)implementation is (mostly) successful. Observe that the average length of paths increases moderately with the number of nodes in the topology (for a given port number) due to the constrained degrees; in most cases, the increment is less than one full hop.

3 STATUS OF PROJECT

The main roadblock we have currently encountered is efficiently computing the bisection bandwidth of network topologies of moderate size. Finding the minimum edge cut that partitions the servers of the network equally is no easy

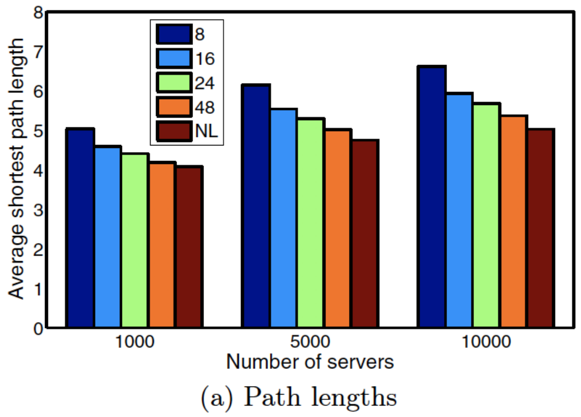


(a)

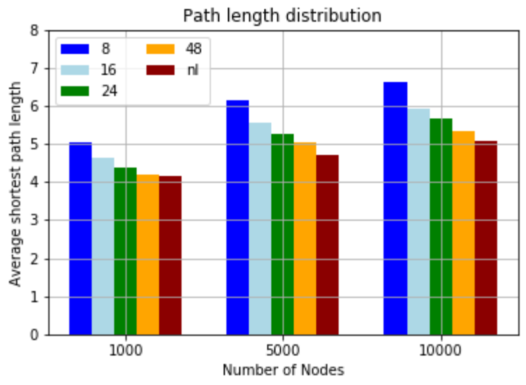


(b)

Figure 1: (a) Original Barabasi-Albert scale-free network w/ 100 nodes, w/o degree limitations (b) Scafida topology w/ 100 nodes, and [10, 20, 10] switches of [4, 8, 16] ports



(a)



(b)

Figure 2: (a) Original Scafida paper average path length distribution graph over 50 simulations. (b) Reproduced figure of average path length distribution over 50 simulations.

task, and currently available optimization algorithms, such as the Kernighan-Lin algorithm, are prohibitively slow on moderately sized networks (e.g. 5000 nodes). Unfortunately, we know of no result that can be used to estimate bisection bandwidth in scale-free networks, even without degree limitations. We are considering several alternatives for an approximate computation, including randomly sampling from the space of valid partitions to obtain an upper bound of the minimum cut from the samples, or computing the bisection bandwidth exactly on much smaller networks (which will likely not be very interesting.)

4 TODO

The plan for the remaining time if as follows; we first need to overcome the intractability of bisection bandwidth computations in order to reproduce figure 4b of the original paper, which plots the cumulative distribution function (CDF) of bisection bandwidths as a function the switch degree limitation. After that, we will explore the topology's fault tolerance by computing the numbers of disjoint paths between servers after switch failures occur, which will be simulated by randomly picking and disabling a fraction of the network's switches. This concludes the main findings of the paper. If time permits, we will proceed to computing metrics for state-of-the-art datacenter topologies using the same equipment numbers and comparing their performance with Scafida's, akin to the original paper's Table 1.

5 APPENDIX: SCAFIDA IMPLEMENTATION

```
# The Scafida Algorithm implementation
def scafida(n_servers, n_ports, n_switches,
     n_switch_ports, m=2):
    V = set()
    for i in range(m):
        V.add(i)

    k = len(n_switches)
    assert k == len(n_switch_ports)
    a_i = Counter()
    degrees = Counter()

    E = set()
    for i in range(m):
        E.add((m, i))
    for i in range(m):
        degrees[i] = 1
    degrees[m] = m

    R = [i for i in range(m)]
    R += [m for i in range(m)]

    n_nodes = n_servers + np.sum(n_switches)
    b = m + 1

    while b < n_nodes:
        V.add(b)
        T = set()

        while len(T) < m:
            v_t = choice(R)
            while True:
                if v_t not in T:
                    break
                v_t = choice(R)
            if degrees[v_t] != n_ports and
                 degrees[v_t] not in n_switch_ports:
                T.add(v_t)
                E.add((v_t, b))

                degrees[v_t] += 1
                degrees[b] += 1
            else:
                nasw = 0
                ntsw = 0

                try:
                    t_i = n_switch_ports.index(v_t)
                except:
                    t_i = 0

                if degrees[v_t] < n_ports:
                    nasw += a_i[0]
                    ntsw += n_servers
                for j in range(k):
                    if degrees[v_t] <
                         n_switch_ports[j]:
                        nasw += a_i[j + 1]
                        ntsw += n_switches[j]

                if nasw < ntsw:
                    a_i[t_i] -= 1
                    a_i[t_i + 1] += 1
                    T.add(v_t)
                    E.add((b, v_t))
                    degrees[v_t] += 1
                    degrees[b] += 1
                else:
                    R = list(filter(lambda a: a !=
                         v_t, R))
        R.extend(T)
        for i in range(m):
            R.append(b)
        b += 1

    return V, E, degrees
```