Service-Concatenation Routing with Applications to Network Functions Virtualization

Shireesh Bhat, George N. Rouskas
Department of Computer Science, North Carolina State University, Raleigh, NC, USA

***Abstract*—Interest in network functions virtualization (NFV) continues to grow due to its perceived benefits to both service providers and users. One of the main challenges in realizing NFV has to do with orchestration of virtual functions deployed in various locations across the network. In this work, we consider the service-concatenation routing problem, where the objective is to construct a path of minimum cost that visits a set of nodes where virtual services are to be applied to the user's traffic in a specific order. We first show that this problem can be modeled as the shortest path tour problem (SPTP) that has been studied in different contexts. We then review and implement a suite of algorithms that use a variety of solution approaches for tackling SPTP, and we also develop a new algorithm. Finally, we carry out a comprehensive experimental evaluation of all algorithms and demonstrate that our algorithm scales well to large problem instances and is suitable for real-time operation as part of the orchestration process in NFV environments.**

I. Introduction

Network functions virtualization (NFV) [1] decouples the network service functionality from the underlying network, compute, and storage resources, and allows communication services to be composed by stitching together functional building blocks that may not be co-located and may be offered by different providers. Interest in NFV has grown dramatically over the past few years due to its perceived benefits to both service providers and the users of these services. One of the main challenges in realizing the potential of NFV relates to orchestration [1], i.e., the process of arrangement and coordination of multiple network services so as to deliver a desired functionality. The role of orchestration in the NFV architecture has been highlighted in previous works [2], [3] that have mostly focused on service abstraction, semantics, and the standardization of the APIs.

Whenever the NFV architecture spans networks operated by multiple distinct/competing network providers and encompasses service components that are geographically apart, orchestration requires (a) a marketplace of services, and (b) specialized routing algorithms. A marketplace [4], [5] acts as a "service commons," a meeting ground where providers publish and advertise the services they offer, and users acquire services based on their requirements and have them instantiated on demand. The concept of a marketplace was also highlighted in our own ChoiceNet project [4] where we envisioned a broader set of network services but focused on introducing a network "economy plane" so as to boost competition among service providers. The marketplace is an essential component in opening up the network infrastructure [2] so as to develop value added services including service composition, fault-tolerance, load balancing, energy minimization, etc., by building upon more primitive virtual network function blocks.

Once the virtual services required by the user have been determined, user traffic must be steered along a path that starts at the source and visits the nodes where the virtual services are implemented, in the order in which they must be applied, before reaching the destination[1]. One variant of this routing problem, referred to as the "node-constrained service chain routing problem" was studied in [6], and was solved by using a layered graph model on which conventional routing algorithms may be applied. Another variant, "service function chaining," was considered in [7], and an algorithm that balances the length of the service function path and the load of service function instances was presented.

In this paper, we consider a general version of the service-concatenation routing problem in NFV environments where the objective is to construct a path that visits a set of nodes where virtual services are to be performed in a specific order. Specifically, our work makes the following contributions:

- We show that the service-concatenation routing problem in NFV may be modeled as a shortest path tour problem (SPTP), a problem that was first studied more than forty years ago in a different context [8], [9].
- We implement all existing algorithms for SPTP that we were able to find in the literature; some of these were originally developed for SPTP, whereas others were developed for related problems and we have modified or extended them to solve SPTP.
- We develop a new algorithm for solving SPTP that outperforms all SPTP algorithms that we are aware of.
- We carry out a comprehensive experimental study to evaluate the performance of all SPTP algorithms and identify their relative merits.

Following the introduction, in Section II we present a model of the NFV marketplace and introduce the general service-concatenation routing problem. In Section III, we define the shortest path tour problem (SPTP) as a model for the service routing problem, and discuss related work. We present and classify existing algorithms for SPTP in Section IV, and we also develop a new algorithm for this problem. We present

This work was supported by the National Science Foundation under Grant CNS-1111088

[1]In this paper we consider point-to-point communication only; multipoint communication will be the subject of future research.

the results of our experimental study in Section V, and we conclude the paper in Section VI.

II. System Model

We consider an NFV environment spanning multiple network domains, possibly administered by distinct network providers. We assume that the NFV environment is shared by a set of service providers who compete against each other to offer network services to users. The network services may include path/routing, data storage, data modification, data analysis, and computation services. This is not an exhaustive list and may potentially be expanded to include services that do not fit into any of these categories, or services that will be developed to satisfy future demands. Service providers utilize NFV abstractions and APIs to deploy multiple instances of each service at strategic points in the network so as to better serve a geographically diverse population of users.

We further assume that the NFV architecture includes a marketplace [4], [5] as an integral component. The marketplace may be thought of as a repository of services and network functions that are available to users. The repository provides APIs for providers to publish (advertise) the services they offer, and for users (or agents acting on their behalf) to obtain lists of service offerings that are relevant to their requirements. To aid users in selecting network services that best match their needs, the orchestration module of NFV uses a planner [10], [11]. In a travel industry analogy, service providers include the airlines, hotels, and rental car companies, whereas travel sites such as Expedia or Travelocity manage marketplaces that include planning and orchestration functions. These functions construct itineraries based on traveler (user) requirements and ensure that users may access seamlessly all the services acquired across the various flight, accommodation, and car rental providers.

An NFV marketplace planner has two main tasks [10], [11]:

- it determines the set of services/virtual functions to meet a user's requirements, and the order in which these services are to be applied to the user's traffic, and
- it constructs a path from source to destination that visits virtual nodes where instances of these services/virtual functions have been deployed.

Some of our earlier work [12], [13] demonstrates a complete lifecycle of the network services on a GENI slice [14], starting with how the network services are described using a semantics language and advertised in a marketplace, followed by how the services are purchased/acquired leading up to their instantiation, and finally how the services are used by the user. In this work, we focus on the second task above, where the objective is to direct user traffic to virtual instances of the service functions that must be applied. Note that it is the concatenation of the services in the order specified that accomplishes the functionality that meets the user's requirements. Therefore, we refer to this problem as "service-concatenation routing," and we define it as:

> *Given an ordering of a set of services, construct a path of minimum cost from source to destination, that traverses nodes where virtual instances of these services reside and may be applied in the specified order.*

The services repository of the NFV marketplace may use any convenient format or data structure to represent the services offered by the various providers. Nevertheless, we assume that the service information stored in the marketplace may be represented in a graph format that makes it possible to apply graph algorithms to solve the service-concatenation routing problem. This graph representation may be maintained internally by the marketplace itself and made available to the planner. Alternatively, the marketplace may provide appropriate APIs that allow external services to repeatedly query the repository so as to construct the graph of services, as we consider in [10]. In the latter case, planners may be offered as competing services external to the marketplace. Either way, we expect that the graph will be highly dynamic in that it will have to be updated every time users acquire new services, or release services they no longer need.

Note that the planner of a travel site takes into consideration flights from multiple airlines, many of which offer competing flights between the same pairs of cities, as well as multiple hotels or rental car facilities within a given city. Similarly, the planner of an NFV marketplace must consider virtual services/functions from multiple providers, including virtual operators who may lease capacity from the same physical infrastructure. Consequently, the planner takes as input a topology that is a superset of the topologies representing the underlying networks. In particular, nodes and edges in the topology represent virtual entities rather than physical ones. For instance, a physical node may include multiple virtual nodes, each virtual node operated by a different service provider deploying a variety of virtual function instances. The graph may also include parallel edges between nodes that represent competing path services. Such a topology is expected to be significantly larger than the underlying physical network topology, hence path finding algorithms must scale to large graph sizes.

As a final note, we assume that the planner has knowledge of the complete topology (graph) of virtual nodes and services, and uses it to solve the service-concatenation routing problem by applying a path finding algorithm. If the NFV architecture is deployed in a software defined networking (SDN) environment, the planner may be implemented as an application of the SDN controller and use the latter's capabilities to construct and maintain this topology. However, our work does not require an SDN environment and applies to any architecture in which the planner has the means to discover and update the complete topology graph.

III. The Shortest Path Tour Problem (SPTP)

Consider the SPTP problem first studied in [8], [9]:

Problem 1 (SPTP): Given

- a graph $G = \{\mathcal{N}, \mathcal{E}\}$ where $\mathcal{N}$ is the set of nodes and $\mathcal{E}$ is the set of edges,
- a source node s and a destination node d, $s, d \in \mathcal{N}$, and

- K non-empty ordered node sets $S_1, S_2, \ldots, S_K$, such that $S_i \subset \mathcal{N}, i = 1, \ldots, K$,

find the shortest path from s to d under the constraint that the path visit one node $n_i \in S_i$ of every set $S_i, i = 1, \ldots, K$, in the given order, i.e., $n_1, n_2, \ldots, n_K$.

We note that whenever each set S_i is a singleton (i.e., $S_i = \{n_i\}, i = 1, \ldots, K$), SPTP reduces to loose source routing as originally specified by the IP protocol [15]. Similarly, whenever there is exactly one node set (i.e., $K = 1$), SPTP becomes similar to anycasting [16].

Recall now the service routing problem we introduced in the previous section, and let K denote the number of virtual services that must be applied to the user's traffic. Without loss of generality, assume that the virtual services are labeled $1, 2, \ldots, K$, in the order in which they must be applied. Finally, let $S_i, i = 1, \ldots, K$, denote the set of nodes where instances of virtual service i reside. Since a path that solves SPTP visits a node for each virtual service, and in the order in which services must be applied, and is the minimum-cost one among all such paths, then it is also a solution to the service-concatenation routing problem defined in the previous section.

Several variants of SPTP have been studied in the literature. The constrained shortest path tour problem (CSPTP) [17] is defined as SPTP with the additional constraint that the path not include repeated edges; whereas SPTP is solvable in polynomial time, this constraints makes the problem NP-Hard. Another variant arises in travel planning applications [18], whereby there exist additional constraints related to the minimum amount of time that a traveler must stay at each node (city). The introduction of such constraints to SPTP converts the problem from polynomial time solvable to pseudo-polynomial [19]. A related problem whose objective is to find the shortest elementary path that visits all nodes in a set S in an arbitrary order (i.e., the input does not include a fixed order on the nodes to be visited) is NP-Complete (NPC) [20]. Relaxing the previous problem to include paths which are not elementary still places the problem in class NPC [20]. Variants of SPTP have also been defined under the class of vertex constrained shortest path (VCSP) problems [21].

IV. Algorithms for SPTP

We now consider the basic SPTP problem we defined in the previous section, and we review and classify all existing algorithms for the problem that we were able to find in the literature. We also present a new algorithm for SPTP that, as we will show later, outperforms earlier algorithms.

A. Path Tour Decomposition

Let us define $S_0 = \{s\}$ and $S_{K+1} = \{d\}$. It has been observed that SPTP may be decomposed into $K + 1$ sub-problems, such that the k-th sub-problem, $k = 0, \ldots, K$, consists of constructing shortest paths from each node in S_k to each node in S_{k+1}.

When SPTP first appeared in the literature [8], [9], it was applied to telephone and transportation networks with large, sparse topologies. Consequently, single source shortest path (SSSP) algorithms were used to solve the SPTP sub-problems. More recently, SPTP has found applications in warehouse management and control of robot motions [22], [23], where the graphs are small but dense. Therefore, researchers and developers have adopted all pair shortest path (APSP) algorithms to solve the sub-problems of SPTP, as these are more efficient for this type of graphs.

A third option for solving each subproblem of SPTP is to apply algorithms for the multiple pairs shortest path (MPSP) problem. MPSP [24]–[26] has a range of applications, from multicommodity network problems to airline network problems, and is concerned with computing shortest paths for a subset of all node pairs in the network. By using algebraic shortest path algorithms [24]–[26], it is possible to reduce significantly unnecessary computations of either APSP algorithms (which construct paths for all node pairs) or SSSP algorithms (which must be executed multiple times, once with each node as the source node).

Therefore, we have three types of decomposition (DC) algorithms for SPTP:

- *DC-APSP:* The algorithm presented in [23] uses APSP to solve each subproblem of SPTP.
- *DC-MPSP:* Although to the best of our knowledge there has been no algorithm for SPTP that uses MPSP for the subproblems, based on our observations above, we have implemented two such algorithms:
  - *DC-MPSP-1:* This implementation uses the MPSP algorithm in [24] to compute each sub-path of the shortest tour between the source and destination nodes.
  - *DC-MPSP-2:* In this version, we apply the MPSP algorithm in [25] at each intermediate stage[2].
- *DC-SSSP:* We have implemented two algorithms that use SSSP:
  - *DC-SSSP-1:* This is a straightforward application of Dijkstra's algorithm to find shortest paths from every node of S_k to every node of S_{k+1}. This algorithm is similar to the one employed in [7] in the context of virtual network function deployment across datacenters, and has also been discussed in [24].
  - *DC-SSSP-2:* The algorithm in [9] also uses SSSP at each stage. It differs from the straightforward algorithm DC-SSSP-1 in that it considers a virtual node v that connects to each node in S_k with zero-cost edges, and applies Dijkstra's algorithm to find the shortest paths from v to each node in S_{k+1}. Hence, it is more efficient since it makes only one call to Dijkstra's algorithm in each stage.

Let $T[]$ be a $(K + 2) \times N$ array such that $T[k, n]$ denotes the cost of the shortest path tour from the source node s to a node $n \in S_k$; this quantity is equal to infinity if

[2]We have also implemented the MPSP algorithm in [26] for the SPTP, but it is significantly less efficient than DC-MPSP-2 and hence we do not consider it in this study.

$n \notin S_k$. Also, let $D(i,j)$ denote the cost of the shortest path from node i to node j in the network graph. Then, the dynamic programming pseudocode of Algorithm 1 describes the operation of a generic decomposition algorithm for the SPTP problem; the only algorithm-specific operation is the computation of the cost $D(i,j)$ of the shortest path between nodes i and j, which may be based on the APSP, MPSP, or SSSP algorithms.

```
Initialization:
$T[k,n] = \infty, k = 0, \dots, K+1, \ \forall\ n \neq s$
$T[0,s] = 0$
for $k = 0, \dots, K+1$ do
    for $i \in S_k$ do
        for $j \in S_{k+1}$ do
            $D(i,j)$ = cost of shortest path from $i$ to $j$
              using APSP, MPSP, or SSSP algorithms
            $T[k+1,j] = \min\{T[k,i]+D(i,j), T[k+1,j]\}$
        end
    end
end
```

Algorithm 1: Generic decomposition algorithm for SPTP

B. Layered Graph Model

A different approach that has been used in the literature for tackling SPTP is to augment the network graph in a way that makes it possible to apply conventional shortest path algorithms to construct the path tour of minimum cost between the source and destination nodes. Specifically, the studies in [6], [22] create a layered graph of $K+1$ layers, each layer consisting of an exact copy of the original network topology. Nodes in adjacent layers are connected with new edges such that any path from the source node (at the lowest layer) to the destination node (at the highest layer) satisfies the path tour constraints. Then, an application of Dijkstra's algorithm is sufficient to determine the minimum-cost path tour.

We have, therefore, implemented this algorithm:

- *LG:* The algorithm described in [22] to solve SPTP on a layered graph; a similar layer graph model is also discussed in [6], although an algorithmic description is not provided.

C. Depth First Tour Search: A New Algorithm for SPTP

We now present a new algorithm for the SPTP problem that eliminates the exploration of nodes in the graph (with respect to computing shortest path to them), whenever such exploration is determined that it will not lead to a better path tour. As a result, our algorithm is quite efficient, and we will present simulation results to demonstrate that it outperforms the algorithms discussed above.

Our algorithm operates similar to Dijkstra's algorithm, but with important enhancements and modifications to make it more efficient and ensure that the SPTP constraints on the path tour are satisfied. The algorithm does not decompose SPTP in subproblems, nor does it employ a layered graph; it operates on the given network graph without modifying it. Specifically, it starts with the source node s and explores nodes using the same criteria as Dijkstra's algorithm, until it reaches the destination node; at that time, the algorithm is guaranteed to have found the shortest path tour that solves the given instance of SPTP. Unlike Dijkstra's algorithm that maintains a single set of encountered nodes (i.e., nodes for which the shortest path from the source has been determined and will not change in the future), our algorithm maintains $K+1$ sets $F_i, i = 1, \dots, K+1$, of encountered nodes: the first K sets $F_i, i = 1, \dots, K$ are associated with reaching nodes in the K sets S_i, respectively, and the last set is for reaching the destination node d. Therefore, a node x may be in one or more sets F_i depending on which part of the tour it has been encountered; for instance, x may be encountered as part of one tour from s to the first set S_1, but it may also be encountered as part of the same or another tour from S_1 to $_2$.

The operation of the algorithm may be summarized as follows:

1) Initially, all the encountered sets are initialized to $\emptyset$ except F_1 which is initialized to contain the source node, i.e., $F_1 = \{s\}, F_i = \emptyset, i = 2, \dots, K+1$.
2) At each iteration l of the algorithm, the node x with the minimum cost is selected. Unlike Dijkstra's algorithm, node x is selected among all the nodes that have not been encountered as part of at least one set F_i. This operation is implemented efficiently by maintaining $K+1$ heaps, each associated with one of the tour stages, and then selecting the minimum cost node among all the heaps. Also note that this feature allows the algorithm to make forward progress towards the destination by continuing towards node set S_{i+1} without waiting for all nodes in node set S_i to be explored first.
3) Our implementation keeps track of which part of the tour node x has been encountered, such that if it is part of the tour from set S_i to set S_{i+1}, then node x will now be included in set F_{i+1}. Also, the cost of each neighbor y of x is updated appropriately (i.e., as in Dijkstra's algorithm), as long as y has not been encountered as part of at least one set F_i.
4) If node x is the last node of some set S_i to be explored (i.e., partial tours that reach all nodes in S_i have now been constructed), then we disregard any partial tours that have only reached nodes in sets $S_{i-1}, \dots, S_1$. Any such partial tours will have higher cost once extended to reach nodes in S_i, hence they cannot be part of the shortest path tour.
5) The algorithm iterates from Step 2 above, until the destination node d has been reached.

Since this algorithm makes progress towards the destination beyond a set S_i without waiting until all nodes of that set have been explored, it bears some similarities with depth first search; hence, we will call our algorithm depth first tour search

(DFTS)[3].

Let $T[]$ be a $(K+1) \times N$ array such that $T[k,n]$ denotes the cost of the shortest path tour from the source node s to a node $n \in \mathcal{N}$. Let C_{xy} be the cost of the directed edge from x to y, where $x, y \in \mathcal{N}$. Algorithm 2 provides a pseudocode description of the DFTS algorithm.

```
Initialization:
$F_1 = \{s\}$
$F_k = \emptyset,\ k = 2, ..., K+1$
$T[k,n] = \infty,\ k = 1, ..., K+1$
$T[1,s] = 0$
$I = 1$
while $d \notin F_{K+1}$ do
    $T[i,w] = \min\{T[i,v] + C_{vw}\}$ s.t.
      $v \in F_i, w \notin F_i, v \notin S_i, i = I, ..., K+1$
    if $w \notin S_i$ then
        $F_i = F_i \cup \{w\}$
    else
        $F_i = F_i \cup \{w\}$
        $F_{i+1} = F_{i+1} \cup \{w\}$
    end
    if $F_i \cap S_i = S_i$ then
        $I = i + 1$
    end
end
```

Algorithm 2: The DFTS algorithm for SPTP

We have the following result regarding the correctness of DFTS.

Theorem 1: For every connected directed graph with non-negative edge costs, DFTS correctly constructs the shortest path tour from the source s to the destination d.

Proof. Let $L[k,n]$ be the true shortest path tour from the source node s to a node $n \in \mathcal{N}$. The proof is by induction and follows the proof of correctness of Dijkstra's algorithm.

Base Case: $T[1,s] = L[1,s] = 0$.

Inductive Hypothesis: All previous found shortest path tours are correct, i.e., $\forall\ n \in \mathcal{N}r : T[k,n] = L[k,n]$.

Current Iteration: We pick an edge (v^*, w^*) which is the minimum cost edge such that $v^* \in F_i$ but $v^* \notin S_i$ and $w^* \notin F_i$, and we let:

$$T[k,w^*] = T[k,v^*] + C_{v^*w^*} = L[k,w^*] + C_{v^*w^*}$$

We distinguish two cases.

Case 1: If $w^* \in S_i$ we add w^* to both F_i and F_{i+1}. Since w^* is present in S_i we are now crossing the frontier of S_i and we need to start exploring nodes in S_{i+1}, so we add w^* to F_{i+1}.

Case 2: If $w^* \notin S_i$ we add w^* to F_i. Since w^* is not present in S_i we are not yet crossing the frontier of S_i corresponding to this node and we need to continue exploring nodes in S_i, so we add w^* to F_i.

[3]Note also that the decomposition algorithms are akin to breadth first search, since they explore all nodes of a set S_i before proceeding to explore nodes in set S_{i+1}

TABLE I
TIME COMPLEXITY

Algorithm	Complexity
DC-APSP [23]	$O(N^3) + O(KM^2)$
DC-MPSP-1 [24]	$O(N^3) + O(KM^2)$
DC-MPSP-2 [25]	$O(N^3) + O(KM^2)$
LG [22]	$O(KN^2) + O(KElog(KN))$
DC-SSSP-1 [7]	$O(2ElogN) + O((K-1)MElogN)$
DC-SSSP-2 [9]	$O((K+1)ElogN)$
DFTS (this work)	$O((K+1)ElogN) + O((K+1)N)$

We now note that every path tour from s to w^* must have cost $\geq L[k,w^*] + C_{v^*w^*}$, therefore this is the cost of any shortest path tour. To show this, let us assume that there is a tour P which has cost $< L[k,w^*] + C_{v^*w^*}$. This tour has to cross from some node explored in F_i to nodes not explored in F_i or nodes not explored in F_{i+1}. If it does, then the edge of cost $C_{v^*w^*}$ selected by the algorithm at this iteration is not the minimum-cost edge, a contradiction. ■

D. Algorithm Complexity

In Table I, we summarize the running time complexity of the seven algorithms we described earlier in this section; we evaluate experimentally the algorithms in the following section.

The DC-APSP [23], DC-MPSP-1 [24], and DC-MPSP-2 [25] algorithms internally use three nested **for** loops to calculate the cost $D(i,j)$ of shortest paths between nodes in adjacent node sets; this computation takes time $O(N^3)$, where N is the number of nodes in the graph, and is shown as the first term of the complexity expression in the top three rows of Table I. The second term in these complexity expressions corresponds to the time it takes to carry out the dynamic programming Algorithm 1. Therefore, APSP and MPSP are efficient when the graph size is not very large, the node sets S_i are large such that it is necessary to compute paths for a substantial fraction of source-destination pairs, and the graph is strongly connected.

The *LG* [22] approach first constructs a modified layered graph which has KN nodes and KE edges; this takes time $O(KN^2)$, where K is the number of layers (node sets). It then applies Dijkstra's algorithm just once on this graph, and the time for this computation is represented by the second term in the appropriate row of Table I.

For the DC-SSSP-1 algorithm, the first expression in the table denotes the use of Dijkstra's algorithm once from s to reach the nodes in S_1, and a second time from d to reach the nodes in S_K, if we reverse the direction the edges. The second expression corresponds to the application of Dijkstra's algorithm a further $(K-1) \times M$ times to find the shortest cost distance from every node in S_i to every node in $S_{i+1}, i = 1, \ldots, K-1$, where $M = \max\{|S_i|\}$. This approach works well for problem instances in which each node set S_i is relatively small compared to the whole graph. DC-SSSP-

2 applies Dijkstra's algorithm $(K+1)$ times for finding the shortest cost path from any node in S_i to every node in $S_{i+1}, i = 0, \ldots, K+1$. DFTS applies Dijkstra's algorithm just once, but every edge may potentially be traversed $(K+1)$ times (first term in the table), and at each iteration it selects the shortest cost edge among $(K+1)$ sets (second term). To fairly compare the last four algorithms, we have implemented them using binary min-heaps, hence the logarithmic terms in the expressions shown in Table I.

V. Experimental Study and Results

We now present simulation results to evaluate the seven algorithms we described in Section IV, namely, DC-APSP, DC-MPSP-1, DC-MPSP-2, DC-SSSP-1, DC-SSSP-2, LG, and DFTS. We evaluate the algorithms on random graphs generated using BRITE [27], a universal topology generator. We obtained undirected graphs by configuring BRITE to generate AS-Level Barabasi models; we then converted these graphs into directed ones that we used in our experiments. In generating random instances for the SPTP problem, we considered the following parameters and varied their values as described below:

- The number N of nodes in the graph was varied from 1000 to 5000 in increments of 1000.
- The average nodal degree Δ of the graph was set to an integer in the range $[2, 5]$.
- The number K of node sets in the tour took integer values in the interval $[1, 4]$; recall that in the service routing problem, K represents the number of services to be applied to the user's traffic.
- The number M of nodes in each node set was varied from 5 to 25 in increments of 5.

Since all algorithms produce the same solution to any instance of SPTP, our evaluation focuses on one metric, running time. We note that the orchestration process in an NFV environment must operate in real time and scale to large network topologies with many services and multiple virtual instances of each service. Hence, the various figures in this section explore the running time of the algorithms as a function of the various parameters listed above. With two exceptions that we discuss shortly, each data point in these figures is the average running time over 10,000 problem instances generated from the stated values of the parameters. All experiments were performed on a HPC cluster that included three processor families, Intel Xeon E5520 (2.27GHz), E5620 (2.40GHz) and E5540 (2.53GHz), all with four cores, each core having 4GB of DRAM and 8KB of cache.

A. Overall Comparison

Figures 1 and 2 plot the running time of the seven algorithms as a function of the number N of nodes in the network. For the problem instances used in these figures, the nodal degree was set to $\Delta = 3$, the number of node sets was $K = 2$, and the number of nodes in each node set was $M = 5$. Our first observation is that the four algorithms (DFTS, LG, and the two DC-SSSP algorithms) shown in Figure 1 take less

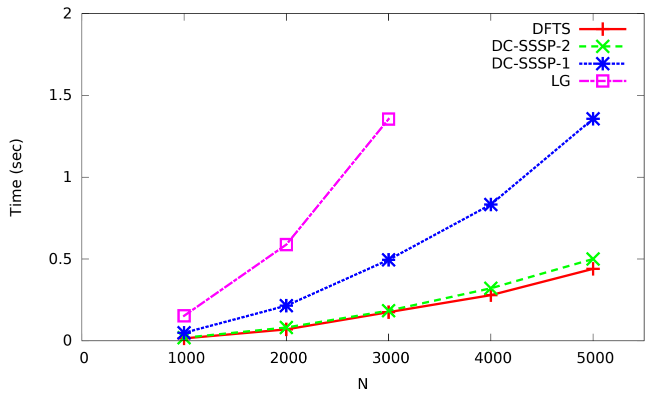

Fig. 1. Running time comparison, most efficient algorithms, $\Delta = 3, K = 2, M = 5$

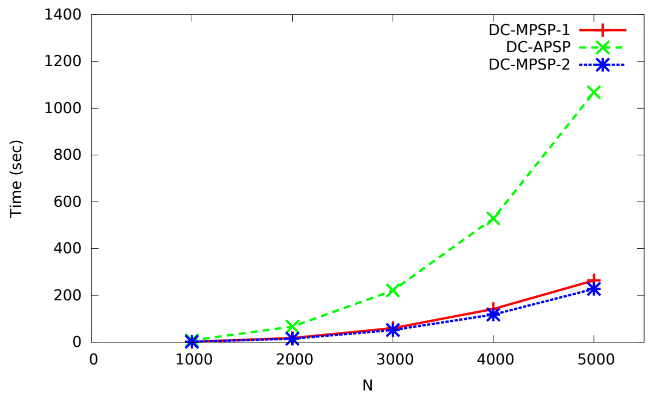

Fig. 2. Running time comparison, least efficient algorithms, $\Delta = 3, K = 2, M = 5$

than two seconds on average to solve these problem instances, whereas the other three (DC-APSP and the two DC-MPSP algorithms), shown in Figure 2 are two-to-three orders of magnitude slower - hence, it was necessary to separate them in a different figure. Furthermore, each data point in Figure 2, as well as in the similar Figure 4 discussed shortly, represents the average of only 50 problem instances, rather than the 10,000 that we used for all other figures. This value was selected as it allowed us to obtain each data point in no more than 24 hours for the largest problem instance considered in these two figures. Another interesting observation from Figure 1 is that no data points are shown for the LG algorithm and networks with more than $N = 3000$ nodes. Recall that the LG algorithm constructs a graph of K layers of the original network topology. Consequently, as the network size grows, it is memory, not running time, that becomes the limiting factor, and we were not able to solve larger instances with the LG algorithm in the HPC cluster available to us.

Figures 3 and 4 are similar to the ones above but present results for instances generated with $\Delta = 5$, $K = 4$, and $M = 25$. Since the problem instances are larger in this case, the running times are higher than the corresponding algorithms in the previous two figures. Similarly, in both sets of figures,

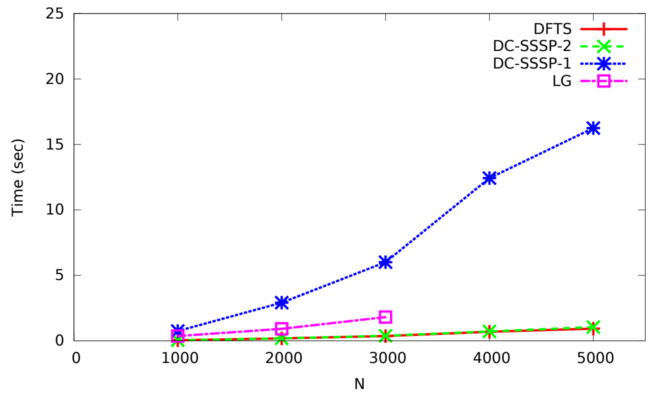

Fig. 3. Running time comparison, most efficient algorithms, $\Delta = 5, K = 4, M = 25$

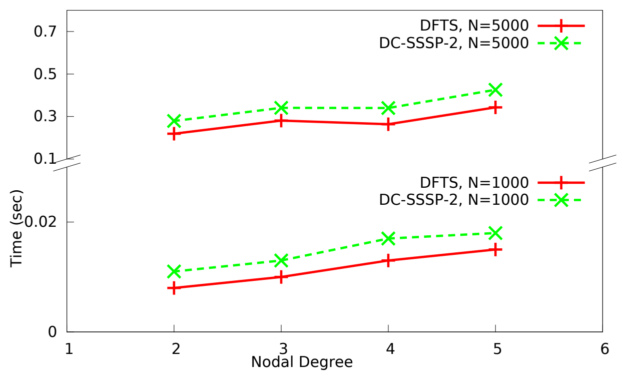

Fig. 5. Running time vs nodal degree, $K = 1, M = 5$

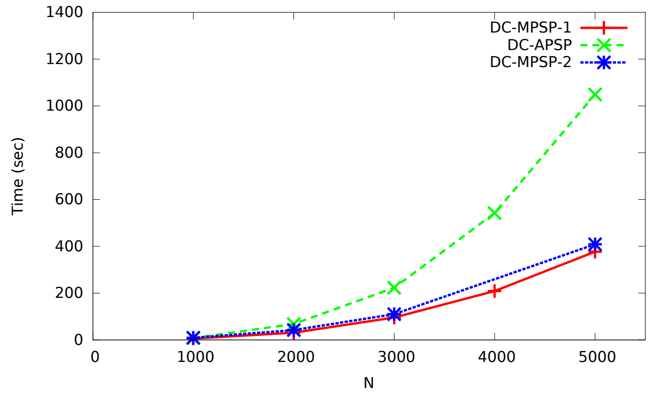

Fig. 4. Running time comparison, least efficient algorithms, $\Delta = 5, K = 4, M = 25$

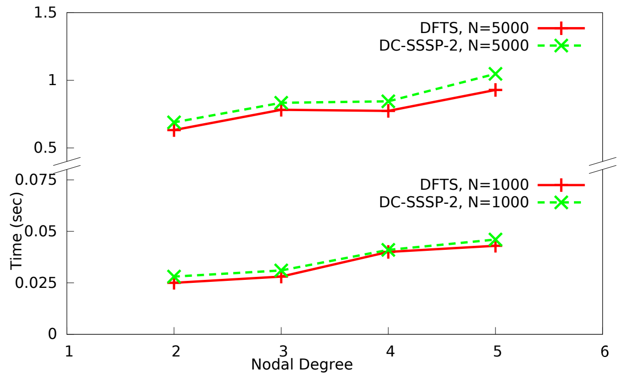

Fig. 6. Running time vs nodal degree, $K = 4, M = 25$

the running time of a particular algorithm increases with the network size N.

From the four figures, it is clear that the three least efficient algorithms (DC-APSP, DC-MPSP-1, DC-MPSP-2) do not scale well and are not appropriate for real-time operation. Also, since LG replicates the network topology K times, its memory requirements become a challenge for larger problem instances. Finally, as Figure 3 illustrates, the DC-SSSP-1 algorithm, which applies Dijkstra's algorithm multiple times at each stage, becomes one order of magnitude slower than the two best algorithms, DFTS and DC-SSSP-2, at larger problem instances considered here.

We have observed the relative behavior illustrated in the four figures above across a wide range of experiments. Therefore, in the remainder of this section we will explore further only the behavior of the two best algorithms, the DC-SSSP-2 algorithm of [9], and the new algorithm we developed, DFTS.

B. *Comparison of DC-SSSP-2 and DFTS*

Let us now investigate the performance of the two algorithms as a function of the parameters Δ, K, and M. Figures 5 and 6 plot the running time of the DC-SSSP-2 and DFTS algorithms by varying the nodal degree Δ of the graph and keeping the values of the other parameters fixed. Each figure includes two sets of plots, one for networks with $N = 1000$ nodes and one with $N = 5000$. In the problem instances of Figure 5, the path tour must visit a node from just one set ($K = 1$) that includes five nodes ($M = 5$); whereas for Figure 6 the problem instances were generated with $K = 4, M = 25$.

As the average nodal degree Δ increases, the size of the network (in terms of the number of edges) grows, hence the running of the two algorithms also increases; however, as we can observe from the two figures, this increase in running time is rather moderate. Similarly, the running time curves for the larger network ($N = 5000$) sit higher than those for the smaller network ($N = 1000$) in the same figure (i.e., for the same values of the other parameters); this behavior is also expected and is due to the increase in network size and consistent with the complexity results in Table I. Also, as K and M increase, the path tour must traverse a larger number of node sets, and there are more options (in terms of number of nodes, M) to be explored, hence the running time values in Figure 6 are higher than for the corresponding curves in Figure 5.

Finally, we make two important observations. First, the running time of either algorithm does not exceed one second

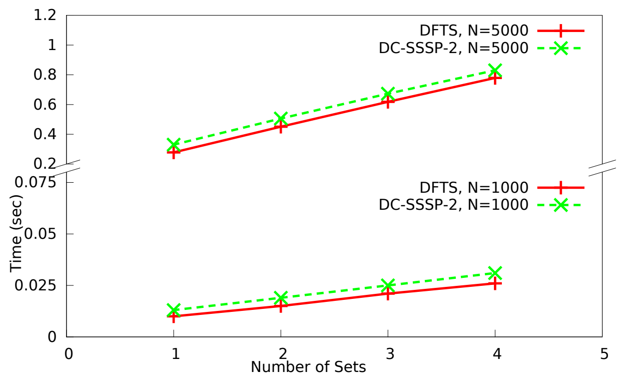

Fig. 7. Running time vs number of sets, $\Delta = 3, M = 15$

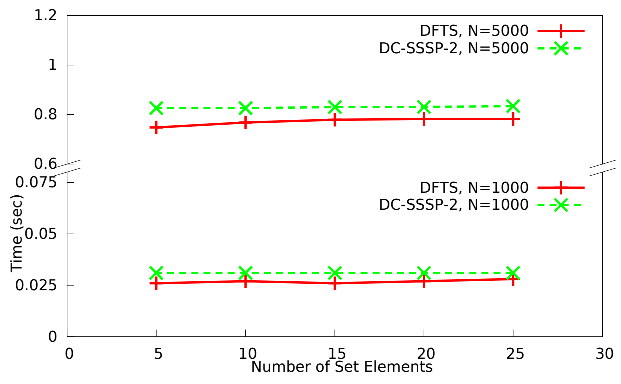

Fig. 9. Running time vs number of set elements, $\Delta = 3, K = 4$

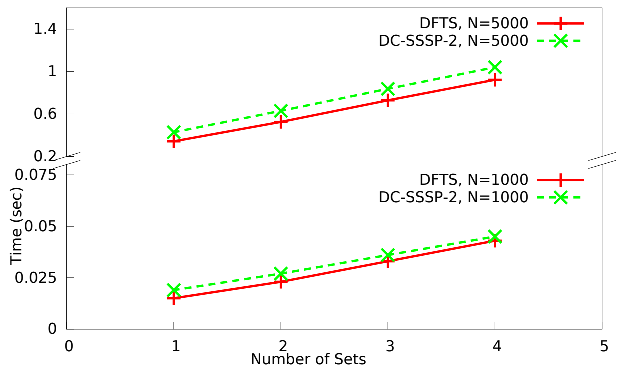

Fig. 8. Running time vs number of sets, $\Delta = 5, M = 15$

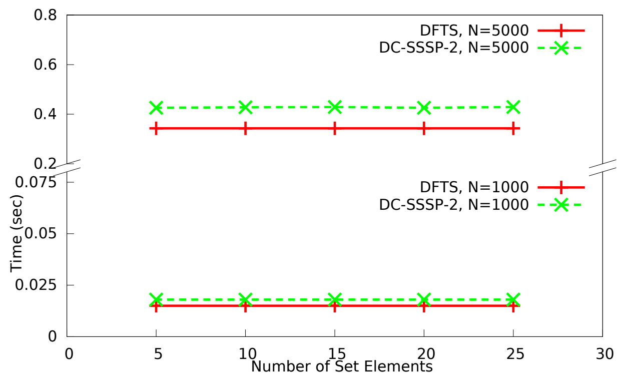

Fig. 10. Running time vs number of set elements, $\Delta = 5, K = 1$

even for the largest of the problem instances we present in the above two figures (i.e., instances with $N = 5000$ nodes, nodal degree $\Delta = 5$, $K = 5$ node sets, and $M = 25$ nodes per set). Therefore, we conclude that these two algorithms scale well and are suitable for real-time applications. Furthermore, our new algorithm, DFTS, consistently outperforms the next best algorithm, DC-SSSP-2, across the range of parameter values that we investigated.

The next two figures, 7 and 8, are similar to the ones we just discussed but plot the running time of the DFTS and DC-SSSP-2 algorithms by varying the number K of sets in a tour while keeping the other parameters fixed. With a larger number of sets, the path tour must traverse more nodes, hence it takes longer time to explore all the options to construct the tour; this intuition is confirmed by the results in the two figures. As before, we also observe that our DFTS algorithm outperforms DC-SSSP-2, and that its running time does not exceed one second, even for the largest instances.

The last pair of figures, 9 and 10, compare the running time of the two algorithms as a function of the number of set elements in a set, with all other parameters fixed. All our observations above regarding the relative and absolute performance of the algorithms are also valid for these sets of results. However, we also observe that the running time of either algorithm is largely insensitive to the size M of the node sets. This is mainly due to the way the two algorithms operate. As we mentioned in Section IV, DC-SSSP-1 applies Dijkstra's algorithm once to find the shortest path from any node in set S_i to any node in set S_{i+1}; as a result, the running time is not affected much by the size of the node sets. Similarly, our DFTS algorithm does not wait until all nodes in a set have been explored, hence, its performance is relatively independent of the set size.

There are 400 unique combinations of the values of parameters N, Δ, K, and M that we considered in our experiments (refer to the top of this section). In Table II, we list the improvement in running time of our DFTS algorithm over the next best algorithm, DC-SSSP-2, for problem instances generated with each of these 400 parameter value combinations. As we can see, our algorithm runs faster than DC-SSSP-2 in all but 10 combinations which are highlighted in bold in the table. Across all problem instances, our algorithm achieves an improvement in running time averaging 13.62%. Overall, these results demonstrate that the new DFTS algorithm exhibits superior performance compared to existing algorithms, it scales well and is suitable for real-time applications.

TABLE II
RUNNING TIME IMPROVEMENT (IN %) OF DFTS RELATIVE TO DC-SSSP-2

		$K = 1$					$K = 2$					$K = 3$					$K = 4$				
N	Δ	$M = 5$	$M = 10$	$M = 15$	$M = 20$	$M = 25$	$M = 5$	$M = 10$	$M = 15$	$M = 20$	$M = 25$	$M = 5$	$M = 10$	$M = 15$	$M = 20$	$M = 25$	$M = 5$	$M = 10$	$M = 15$	$M = 20$	$M = 25$
1000	2	26.51	23.83	28.70	31.73	30.56	25.37	19.48	20.74	24.23	20.91	23.00	20.14	18.45	100.00	16.23	16.33	14.46	**-1.16**	14.13	11.51
	3	21.85	24.77	24.19	25.16	25.56	19.77	13.38	19.89	8.75	9.38	17.57	16.71	15.82	15.94	19.10	14.70	6.68	17.01	14.30	11.20
	4	22.03	15.69	20.84	13.91	18.73	11.79	8.30	8.43	13.58	8.97	9.51	**-0.13**	9.29	9.90	9.92	**-5.41**	2.86	**-10.65**	**-8.03**	3.11
	5	17.51	18.58	16.87	19.77	19.38	13.94	7.42	15.45	6.03	10.21	11.72	7.78	10.62	7.53	7.06	7.56	6.50	6.04	5.90	5.04
2000	2	24.87	26.03	29.13	27.82	29.09	19.63	23.31	19.02	24.36	20.83	16.01	15.63	14.10	16.02	15.94	14.25	11.36	13.07	11.67	14.21
	3	18.65	20.32	9.57	15.60	18.02	14.10	11.90	12.01	12.87	12.50	12.40	14.22	10.72	8.98	9.34	10.22	7.42	5.41	6.13	6.27
	4	17.54	19.41	6.00	15.43	16.43	8.67	10.89	12.44	12.22	12.51	10.87	8.41	8.97	8.47	**-0.76**	9.95	6.52	10.39	5.36	5.91
	5	15.69	14.98	8.54	16.75	17.56	1.76	12.07	10.95	11.61	12.08	14.10	12.37	6.81	8.90	9.24	7.83	5.17	6.76	**-5.60**	**-3.17**
3000	2	18.35	20.83	19.72	20.10	20.72	7.49	14.93	11.72	14.73	15.55	14.96	11.06	10.60	11.72	11.30	12.98	9.59	8.94	8.21	21.75
	3	20.49	17.35	17.24	25.33	18.74	4.67	26.96	11.08	11.44	16.87	10.77	9.37	8.29	8.51	8.61	10.04	6.73	6.55	6.73	4.43
	4	15.48	15.43	15.42	14.83	14.98	3.44	9.69	10.48	8.61	9.72	0.37	7.49	7.81	7.47	10.65	8.55	5.92	10.09	5.30	5.21
	5	16.20	15.08	15.06	20.35	17.05	11.14	10.39	7.08	14.00	14.52	14.21	11.39	10.98	10.39	5.91	21.84	**-5.53**	3.66	**-4.29**	4.09
4000	2	21.25	20.17	14.16	19.16	23.08	21.27	15.02	14.07	16.62	16.40	14.15	11.72	10.44	11.76	12.63	13.20	10.40	7.30	12.23	8.03
	3	17.86	17.81	17.10	8.56	17.61	12.88	12.30	11.71	10.61	11.97	12.07	7.87	8.27	8.37	0.49	9.91	6.88	6.66	6.45	10.68
	4	23.93	18.24	21.14	21.59	21.13	15.20	15.37	14.10	15.36	10.70	9.18	10.67	11.65	15.33	12.89	10.63	10.96	21.37	9.29	14.22
	5	18.70	12.95	14.80	13.55	14.50	14.01	8.73	6.87	9.42	9.47	7.79	19.59	19.75	4.23	6.42	5.24	5.09	8.95	4.86	4.38
5000	2	21.44	21.83	23.67	24.06	24.58	18.38	18.02	18.51	18.95	18.42	15.14	13.71	8.31	15.52	14.16	14.14	12.19	12.24	12.15	8.32
	3	17.39	14.04	15.66	16.15	16.88	12.16	10.72	10.97	9.74	10.66	10.73	8.26	8.16	7.70	7.34	9.50	7.06	6.14	5.87	6.23
	4	22.16	21.45	22.13	22.43	22.87	17.62	13.77	14.53	16.05	16.01	14.00	11.13	11.80	11.27	11.02	13.54	9.01	8.93	8.57	8.45
	5	19.58	24.37	20.19	20.89	21.06	15.95	15.97	16.45	16.96	16.26	13.31	12.41	12.98	14.69	14.08	11.53	9.69	11.33	10.85	11.37

VI. CONCLUDING REMARKS

The service-concatenation routing problem arises as an integral part of the orchestration process in NFV architectures. We have shown that service-concatenation routing is equivalent to the shortest path tour problem (SPTP). Most existing algorithmic approaches to SPTP work well only for specific classes of problem instances, and do not scale well to the large instances that arise in NFV applications. We have developed a new algorithm that applies several novel modifications to Dijkstra's algorithm to construct the shortest path tour efficiently. Our experimental study has demonstrated that our algorithm scales well to large instances and is appropriate for real-time NFV applications across a wide range of graphs.

REFERENCES

[1] SDN and OpenFlow World Congress. Network Function Virtualization, updated white paper. https: //portal.etsi.org/nfv/nfv_white_paper2.pdf. October 2013.

[2] S. Palkar *et al.* E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM.

[3] A. Gember *et al.* Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR*, abs/1305.0209, 2013.

[4] T. Wolf *et al.* Choicenet: Toward an economy plane for the internet. *SIGCOMM Comput. Commun. Rev.*, 44(3):58–65, July 2014.

[5] G. Xilouris *et al.* T-nova: A marketplace for virtualized network functions. *Proc. EuCNC 2014*, June 2014.

[6] A. Dwaraki and T. Wolf. Adaptive service-chain routing for virtual network functions in software-defined networks. *Proc. of ACM Hot-MIddlebox 2016*, pages 32–37, New York, NY, USA.

[7] A. M. Medhat *et al.* Near optimal service function path instantiation in a multi-datacenter environment. *Proc. CNSM 2015*, pp. 336–341, 2015.

[8] C. P. Bajaj. Some constrained shortest-route problems. *Unternehmensforschung*, 15(1):287–301, 1971.

[9] A. Kershenbaum *et al.* Constrained routing in large sparse networks. *Proc. IEEE ICC," pp. 38.14-38.18, Philadelphia, PA*, 1976.

[10] S. Bhat and G. N. Rouskas. On Routing Algorithms for Open Marketplaces of Path Services. *Proc. of IEEE ICC 2016*, May 2016.

[11] X. Huang *et al.* Automated service composition and routing in networks with data-path services. *Proc. of ICCCN 2010* Aug 2010.

[12] S. Bhat, R. Udechukwu, R. Dutta, and G. N. Rouskas. Inception to Application: A GENI based prototype of an Open Marketplace for Network Services. *IEEE Infocom Workshops*, April 2016.

[13] R. Udechukwu, S. Bhat, R. Dutta, and G. N. Rouskas. Language of choice: On embedding choice-related semantics in a realizable protocol. *Proc. of 37th IEEE Sarnoff Symposium*, Sep 2016.

[14] M. Berman *et al.* Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5 – 23, 2014. Special issue on Future Internet Testbeds Part I.

[15] Internet Engineering Task Force. *RFC 791 Internet Protocol - DARPA Inernet Programm, Protocol Specification*, September 1981.

[16] Internet Engineering Task Force. *RFC 1546 – Host Anycasting Service*, November 1993.

[17] D. Ferone *et al.* The constrained shortest path tour problem. *Computers & Operations Research*, 74:64 – 77, 2016.

[18] J-F. Brub, J-Y. Potvin, and J. Vaucher. Time-dependent shortest paths through a fixed sequence of nodes: application to a travel planning problem. *Computers and Operations Research*, 33(6):1838 – 1856, 2006.

[19] S. Irnich and G. Desaulniers. *Shortest Path Problems with Resource Constraints*, pages 33–65. Springer US, Boston, MA, 2005.

[20] T. Ibaraki. Algorithms for obtaining shortest paths visiting specified nodes. *SIAM Review*, 15(2):309–317, 1973.

[21] J. E. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379–394, 1989.

[22] P. Festa. Complexity analysis and optimization of the shortest path tour problem. *Optimization Letters*, 6(1):163–175, 2012.

[23] P. Festa *et al.* Solving the shortest path tour problem. *European Journal of Operational Research*, 230(3):464 – 474, 2013.

[24] I-L. Wang, E. L. Johnson, and J. S. Sokol. A multiple pairs shortest path algorithm. *Transportation Science*, 39(4):465–476, 2005.

[25] B. A. Carre. A matrix factorization method for finding optimal paths through networks. *Computer Aided Design*, 51(4):388–397, 1969.

[26] B. A. Carre. An elimination method for minimal-cost network flow problems. *J.K.Reid (Ed.), Large sparse sets of linear equations (Proc. I.M.A. Conf., Oxford, 1970). Academic Press: London*, 1971.

[27] A. Medina, I. Matta, and J. Byers. Brite: A flexible generator of internet topologies. Tech. report, Boston University, Boston, MA, USA, 2000.