NETEMBED: A Network Resource Mapping Service for Distributed Applications[†]

JORGE LONDOÑO[‡] AZER BESTAVROS

jmlon@cs.bu.edu best@cs.bu.edu

Computer Science Department

Boston University

Abstract

Emerging configurable infrastructures such as large-scale overlays and grids, distributed testbeds, and sensor networks comprise diverse sets of available computing resources (e.g., CPU and OS capabilities and memory constraints) and network conditions (e.g., link delay, bandwidth, loss rate, and jitter) whose characteristics are both complex and time-varying. At the same time, distributed applications to be deployed on these infrastructures exhibit increasingly complex constraints and requirements on resources they wish to utilize. Examples include selecting nodes and links to schedule an overlay multicast file transfer across the Grid, or embedding a network experiment with specific resource constraints in a distributed testbed such as PlanetLab. Thus, a common problem facing the efficient deployment of distributed applications on these infrastructures is that of "mapping" application-level requirements onto the network in such a manner that the requirements of the application are realized, assuming that the underlying characteristics of the network are known. We refer to this problem as the *network embedding problem*. In this paper, we propose a new approach to tackle this combinatorially-hard problem. Thanks to a number of heuristics, our approach greatly improves performance and scalability over previously existing techniques. It does so by pruning large portions of the search space *without* overlooking any valid embedding. We present a construction that allows a compact representation of candidate embeddings, which is maintained by carefully controlling the order via which candidate mappings are inserted and invalid mappings are removed. We present an implementation of our proposed technique, which we call NETEMBED – a service that identify feasible mappings of a virtual network configuration (the query network) to an existing real infrastructure or testbed (the hosting network). We present results of extensive performance evaluation experiments of NETEMBED using several combinations of real and synthetic network topologies. Our results show that our NETEMBED service is quite effective in identifying one (or all) possible embeddings for quite sizable queries and hosting networks – much larger than what any of the existing techniques or services are able to handle.

Index Terms

Constrained Network Mapping; Overlay Networks; Resource Allocation; Grid Computing and Services.

I. INTRODUCTION

Motivation: An emerging trend in distributed system design is the use of configurable networks, in which network resources may be named, acquired, and manipulated by applications and users in a much more flexible manner than what is possible using current, standard Internet services. Using today's Internet services, such as DNS, an application is able to barely resolve (or map) a name to an IP address, but not much more. We envision more flexible mapping services that allow an application to name resources in a

[†] This work is supported in part by NSF CNS Cybertrust Award #0524477, NSF CNS NeTS Award #0520166, NSF CNS ITR Award #0205294, and EIA RI Award 0202067.

[‡] Supported in part by the Universidad Pontificia Bolivariana and COLCIENCIAS–Instituto Colombiano para el Desarrollo de la Ciencia y la Tecnología "Francisco José de Caldas".

much more expressive fashion – for instance by specifying a desirable topological configuration of nodes and links, and by constraining the attributes of such nodes and links.

One canonical example of a configurable network is that of an overlay network of end-systems. Such overlays are increasingly used as targets of choice for new distributed applications, including large-scale data management systems, *e.g.*, distributed data stores [1], [2], content delivery networks, [3], [4], [5], [6], [7], [8], and specialized resource reservation/allocation services supporting the operation of grid computing [9]. Applications running on overlays require services that allow them to flexibly make content placement decisions, routing decisions, and other decisions regarding resource selection.

Another canonical example of a configurable network is that of network testbeds such as Emulab, Netbed [10] and PlanetLab [11], as well as the envisioned GENI testbed [12]. The development of these network testbeds is motivated by the necessity of having more realistic experimental environments, where new protocols and services can be tested under conditions as close as possible to those found in real networks. Users of (and applications running atop) such testbeds are allowed considerable flexibility in selecting the sets of resources used in conducting a distributed experiment.

Challenge: A common challenge facing the wide adoption of configurable networks as hosting infrastructures for novel distributed applications is the network mapping problem [13]: Given the requirements of an application and the characteristics of the underlying hosting network, find (if possible) a mapping between the nodes and links requested by the application and the nodes and links of the hosting underlying network. Since the topology requested by an application is typically much smaller than the topology of the real network, the network mapping problem is better viewed as a *network embedding problem* – that of embedding the *virtual network* requested by an application into the *real network* of the underlying infrastructure.

At its heart, the network embedding problem is a combinatorial search problem. Since any mapping (of nodes and links) from the virtual network to the real network could potentially be a solution, in principle, all permutations have to be examined if we want to find one (or all) feasible solutions. In practice, however, the more constrained the virtual topology is, the less the number of links/nodes that will satisfy some of the constraints. Such simple observation allows us to prune significant portions of the search space, turning what would be otherwise a computationally-impractical problem into a problem solvable for networks of sizes at least as large as the networks used in current generation overlays and testbeds, and those in the foreseeable future.

Contribution: In this paper, we propose an efficient approach to tackle the network embedding problem. Thanks to a number of heuristics, our approach greatly improves performance and scalability over previously existing techniques. It does so by pruning large portions of the search space *without* overlooking any valid embeddings: If an embedding is possible, our approach well find it, if given enough time. We present a construction that allows a compact representation of candidate embeddings, which is maintained by carefully controlling the order via which candidate mappings are inserted and invalid mappings are removed. We present an implementation of our proposed technique, which we call NETEMBED – a service that identifies feasible embeddings of a virtual network configuration (the query network) onto an existing real infrastructure or testbed (the hosting network). We present results of extensive performance evaluation experiments of NETEMBED using several combinations of real and synthetic network topologies. Our results show that our NETEMBED service is quite effective in identifying one (or all) possible embeddings for quite sizable virtual and hosting networks – much larger than what any of the existing techniques or services are able to handle.

II. Related Work

Published works investigating solutions to the network embedding problem can be broadly classified as casting the problem as an *optimization problem*, or as a *constraint satisfaction problem.*[1]

Network Embedding as an Optimization Problem: Viewed as an optimization problem, solving the network embedding problem reduces to that of finding a mapping that minimizes a well-defined cost metric, or maximizes some objective function. An example of this is the `assign` method [13] developed in support of the operation of the Emulab/Netbed testbed, where the goal of the network embedding is to maximize the potential for deploying further/future experiments on a fixed infrastructure. Using `assign`, resources are classified into groups of identical characteristics, and once assigned the testbed synthesizes the network according to the requested parameters. Scarce resources must then be assigned greedily to increase the chances of having enough space to support future experiments, and simulated annealing is used to optimize this assignment. In [10], another tool called `wanassign`, which uses genetic algorithms, was developed and evaluated for optimizing the mapping of resources in a wide-area distributed setting. The performance evaluation results reported in [10] for `wanassign` considered only small networks, with up to 16 nodes. In [14] the same tool was evaluated for larger (wireless) networks of up to 160 nodes, exposing the main scalability limitations of this approach, given that the time it takes to solve these cases is in the range of *tens to hundreds of minutes*. Scalability aside, another major weakness of techniques that resort to heuristics such as simulated annealing and genetic algorithms is the fact that such techniques give no guarantee of convergence, let alone optimality.

The algorithm by Zhu et al [15] is another example of works that cast network embedding as an optimization problem. This work uses a stress metric on links and nodes to make sure that the resources allocated to each virtual network cause minimal interference with existing virtual networks, with the goal of maximizing the number of virtual networks that can be accommodated within the shared infrastructure. As stated in [15], this algorithm can be extended to the constrained version of the problem by filtering out infeasible assignments. There are no claims about scalability in terms of the size of the virtual and hosting networks. We also note that the approach presented in [15] is particularly tailored to closed networks since it requires an accounting of the stress metric on every real link in the network. This requires precise knowledge and control of the real topology, which obviously is not the case in open networks such as the Internet, or overlay networks such as PlanetLab.

Network Embedding as a Constraint Satisfaction Problem: Viewed as a constraint satisfaction problem, solving the network embedding problem reduces to that of finding a mapping that satisfies a set of topological and parametric constraints. An example of this is the method proposed by Considine and Byers [16]. Here, a constraint matrix describes the requested virtual topology and an inference matrix describes the real network. Although the problem is NP-complete, a brute-force approach coupled with appropriate pruning techniques finds results in reasonable times. In [16], two pruning techniques are proposed to reduce the search space: The first is to prune partial mappings if they cannot be embedded, and the second is to use automorphism to represent multiple equivalent mappings efficiently using a single mapping. While the use of this approach proved to be quite efficient in finding small-sized cliques (with associated link constraints) within a large topology such as PlanetLab, its applicability and scalability for embedding more general structures—namely, embedding arbitrary topologies with constraints over multiple metrics—is not evident.

[1] It is important to note that these two approaches are not exclusive. In particular, the solution to a constraint satisfaction problem may yield multiple feasible embeddings, in which case the embedding of choice would be the one that minimizes a specific cost metric, for example.

Network Embedding as an Integrated Service: The network embedding approaches described so far have focused on the algorithmic/heuristic underpinnings of the combinatorial search problem at hand, without much attention to other important dimensions of the problem – specifically, how to capture the state of a real open network (such as Internet grids or PlanetLab) over time, and how to allow for an expressive specification of the virtual network to be embedded therein.

SWORD [17] is an example of a more integrated approach to solving the network embedding problem. SWORD was developed specifically for PlanetLab. It integrates the several components necessary to provide a mapping service, including: (1) a monitoring infrastructure that maintains an up-to-date image of the state of all the network nodes, (2) a query interface where the user makes a request using a query language specifically designed for SWORD, (3) a query processor which selects candidate nodes, and (4) matcher and optimizer algorithms that select the optimal matching among a set of candidate nodes. SWORD's query language incorporates the notion of optimal matching (zero cost), suboptimal matching regions (cost increases linearly as matching distance increases), and infinity cost regions, where there is no match. SWORD's mapping algorithm explores the non-infinity-penalty assignment space in two phases. In the first phase, it explores matches of groups of nodes to sites returning a list of candidate matchings for each group. In the second phase, it matches the inter-group requirements by trying all possible combinations of group assignments. To improve scalability, SWORD uses a number of heuristics. One such heuristic is to subject each one of the two phases to a timeout. If the algorithm is not finished by the given time, it returns the best answer available so far. Another heuristic is to prune the high penalty candidate groups from the first stage before running the second stage. Here a number of pruning approaches are considered, such as keeping the top half of candidates, the top five candidates, or only the very first candidate. Even if one accepts the arbitrary nature of these heuristics, it is clear that using such measures will necessary compromise completeness (or optimality) in the sense that these heuristics may well result in false negatives, *i.e.,* the algorithm returns a "no match" answer *before* the timeout period expires, whereas in reality a feasible embedding may well exist.

Our Work in Context: In this paper we present three new algorithms that follow the constraint satisfaction approach, but leave open the possibility of using optimizations in a later stage. Our algorithms – and the NETEMBED service we develop based thereupon – are differentiated from prior work along three dimensions.

(1) NETEMBED is designed to ensure completeness and correctness. Using an ordered and thorough exploration of the search space, it prunes *only* those regions of the search space that cannot include any feasible embeddings. Thus, the algorithms implemented in NETEMBED guarantee completeness (if an embedding exists, it will be found) and correctness (if an embedding is returned, it is feasible).

(2) NETEMBED is designed to allow trading off completeness for timely convergence. Converging to all feasible embeddings in a reasonable amount of time may not be possible for extremely large networks or under-constrained queries. Our design allows users and applications to tradeoff completeness for timely convergence by allowing only a subset of the feasible embeddings to be returned within a given time constraint (timeout).

(3) NETEMBED can be used with open networks. It can be deployed without requiring changes to an existing infrastructure. The only requirement is to have a service that provides a characterization of the infrastructure. In our evaluation we use such a service (available for PlanetLab). Moreover, NETEMBED can be integrated as a service and implemented in a distributed fashion, allowing it to cope with complex resource requirement queries that distributed applications demand, such as those mentioned above.

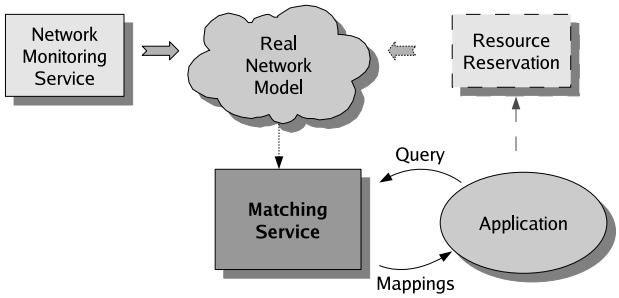


Fig. 1. Architecture of the NETEMBED service, showing its basic components.

III. THE NETEMBED SERVICE MODEL

An embedding service could be thought of as a component of a large distributed system that other applications could use as a primitive for the realization of their resource needs. The following are example application scenarios:

- A dynamic multicast service, where an overlay distribution tree must be configured subject to a set of constraints so that some QoS requirements are satisfied.
- A peer-to-peer system, where location, bandwidth and delay of a subset of nodes, such as "directory nodes" (*e.g.*, the nodes of a distributed hash table) play an important role in the performance of the lookup service and/or the overall performance of the whole system.
- A network monitoring system, where the health of the network could be monitored by a subset of nodes satisfying some connectivity and location constraints.
- A sensor network in which it is desirable to locate a subset of sensors that possess certain capabilities and satisfy some resource, location, and/or network connectivity constraints.
- A grid application that needs to allocate a subset of nodes with certain capabilities and some connectivity requirements between them.

In such cases, it makes sense to have a resource mapping service where applications can submit their resource requirement specifications, and in response get a list of possible resource assignments. Such a service would involve the key components illustrated in Figure 1 and described below:

1. A model of the real network that characterizes the resources available. Such model could be maintained either by a monitoring service, a resource manager, or a combination of both.
2. The mapping service itself, where applications would submit their queries and get a list of possible mappings. An interactive service would facilitate the adjustment (negotiation) of the requirements if the query cannot be satisfied, or allow it to adjust the mapping dynamically, as the application needs change. The service can operate in a distributed fashion simply by keeping an up-to-date copy of the model on each server.
3. Optionally, if a resource reservation system is in place, applications would allocate the selected mapping and the network model would be adjusted accordingly.

As described, the NETEMBED service is focused on finding "feasible" mappings as opposed to finding an "optimal" mapping. As we alluded in Section II, treating the network embedding as an optimization problem involves the identification of a utility/cost (or objective) function which must be maximized/minimized. We note that such objective functions are intimately tied to the application. As such we believe that such considerations are outside the scope of the NETEMBED service.

IV. BASIC DEFINITIONS AND TERMINOLOGY

In this section, we provide the basic definitions for terms we have alluded to earlier and which we use throughout this paper.

We use the term *Hosting Network* to refer to the target of an embedding service. A hosting network (*e.g.,* Internet or PlanetLab) is described by a graph $R =< V, E >$ and a characterization of its links and nodes, which may include measured metrics represented either as numeric values or ranges, and categorical classes such as *"Link* (n_1, n_2) *is 802.11g"* or *"node* n_1 *is linux-2.6"*, for example.

We use the term *Query Network* to refer to the network that needs to be embedded into the hosting network. A query network is given by the graph $Q =< V, E >$ and a characterization of its links and nodes, which represent constraints on any feasible embedding. As with the hosting network, such constraints may be numerical or categorical in nature.

We use the term *Mapping* to refer to a one-to-one (injective) function $m : Q \rightarrow R$, such that for all query network nodes $n_Q \in Q$, $n_R = m(n_Q)$ is the corresponding node in the hosting network, and all node and edge constraints are satisfied by such mapping. To simplify notation, we will use $q \rightarrow r$ to indicate that node q maps to node r. In principle, if N_Q is the number of nodes in Q and N_R is the number of nodes of R, then any permutation of N_Q elements of N_R could be a mapping, making the search space very large even for small values of N_Q and N_R.

We use the term *Constraint Expression* to refer to a boolean expression that specifies additional relationships that must be preserved by the mapping function. For example we may be interested in a mapping that restricts the average delay between nodes within a percentage range, or that certain particular nodes get bound to physical nodes with certain attributes, say operating system, processor type, speed, *etc.* Such constraints take the form of a boolean expression relating query network nodes/links to hosting network nodes/links.[2]

We use the term *Embedding Service* to refer to a system that takes as input the description of a hosting network and the specification of a query network and returns as output a (possibly empty) set of mappings from the latter to the former. Typically, an embedding service would be associated with a single hosting network that constitutes a "real" infrastructure and would be used by users or applications to facilitate the identification of resources that could be used to instantiate a "virtual" network that satisfies the constraints specified in the query network.

In this paper, we use the terms "hosting network", "query network", and "mapping" interchangeably with the terms "real network", "virtual network", and "embedding" respectively. The latter set of terms are typically used when referring to an embedding service tied to a specific infrastructure or testbed.

V. NETEMBED MAPPING ALGORITHMS

In this section, we present the three basic algorithms used in NETEMBED to search for feasible mappings of query network nodes/links to hosting network nodes/links.

A. Exhaustive Search with Constraint Filtering (ECF)

This algorithm uses a depth-first search of the permutations tree (see illustration in Figure 2), pruning any branch as soon as that branch proves infeasible. Any leaf node found during the search indentifies

[2] In NETEMBED, we use a simple constraint expression language, which we describe later in Section VI.

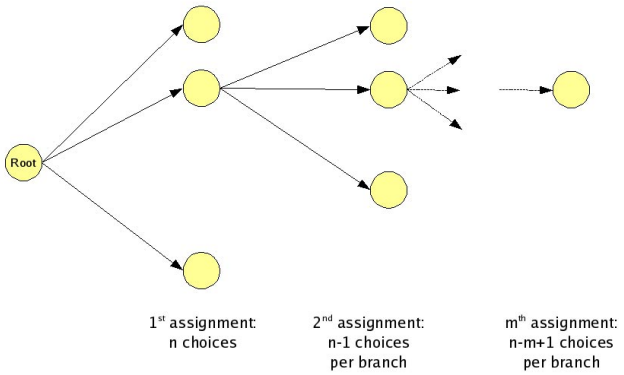


Fig. 2. Illustration of the permutation tree that defines the search space.

a feasible mapping of the query (given by the assignment of each of the nodes in the path from this leaf node to the root). The time taken by the depth-first search is proportional to the number of visited nodes. Lemma 1 gives a criterion that minimizes the total number of nodes that must be searched in the tree.

***Lemma** 1 (Minimum Size of the Permutations Tree)* The total number of nodes in the permutations tree for a search algorithm is minimized if the algorithm examines the query nodes in an ascending order based on the number of candidate mappings for each query node.

Proof: See the Appendix for a sketch of the proof. ■

The above Lemma implies that it is advantageous for the degree of nodes closer to the root of the permutations tree to be as small as possible. Thus, by applying the constraint expression our ECF algorithm determines the number of possible mappings for each virtual node and sorts them in increasing order in a list L_S.

In addition to the above observation, we also note that given the current assignments for $q_1, ..., q_{i-1}$, if q_i has edges with any of its predecessors, the number of choices is reduced even more because these edges have to be preserved.

During the first stage of the ECF algorithm, the constraint expression is applied to each possible pair of virtual and real edges. As illustrated in figure 3, this produces a list of candidate mappings per edge of the form:

$$\{(q_1 \rightarrow r_1, q_2 \rightarrow r_2), ...\}$$

ECF stores this mapping in a data structure that provides the candidates for the second stage. The data structure is a sparse 3D-matrix F that we refer to as the *filter matrix*. Each cell in F has coordinates (v, r, v_s) and contains the set of candidate mappings for v_s, when v is mapped into r. Therefore, each edge matching adds one element in two cells[3] :

$$(q_1, r_1, q_2) \leftarrow r_2 \qquad (q_2, r_2, q_1) \leftarrow r_1$$

Conversely, when there is no match, ECF keeps these results in a second filter $\overline{F}$, constructed in exactly the same way. $\overline{F}$ will then be useful in restricting the set of candidates given the current partial mapping.

[3] For the undirected case. If the network is represented as a directed graph only one of the sets is updated

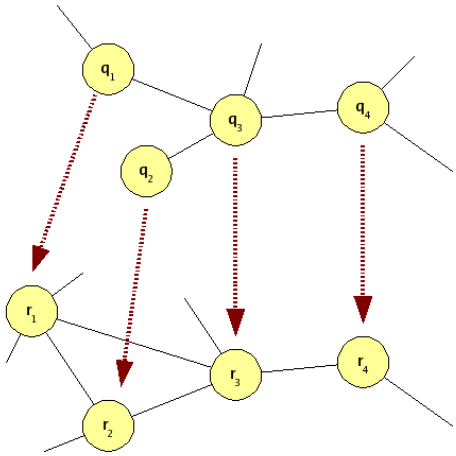


Fig. 3. Illustration of the mappings from a query network to a hosting network.

The ECF algorithm then proceeds as follows:

(1) Pick the first virtual node v_s from the sorted list L_S, i.e. the one with the least number of candidates. Being the first, it can be chosen from:

$$\bigcup_{all\ v \in N_Q, r \in N_R} F[v, r, v_s] \tag{1}$$

(2) For subsequent nodes, say node v_i, choose the mappings from the intersection of candidates for all previous nodes v_j that have an edge with v_i and that do not violate any of the constraints. Real nodes that have been already assigned cannot be considered. Expression (2) gives the set of candidate nodes[4] . This process guarantees that each additional node mapping is consistent with the topology and the established constraints.

$$\left(\bigcap_{all\ j<i | (v_j, v_i) \in E_Q \wedge (v_i, v_j) \in E_Q} F[v_j, r_j, v_i] \right) - \left(\bigcup_{all\ j<i | (v_j, v_i) \in E_Q \wedge (v_i, v_j) \in E_Q} \overline{F}[v_j, r_j, v_i] \right) - \{r_1, ..., r_{i-1}\} \tag{2}$$

Once N_Q mappings have been found using the above process, then by induction we get a valid mapping for the whole query. The above process can be efficiently implemented. The complete algorithm incorporating these filters is given in Figure 4.

Correctness and Completeness of ECF: ECF does a depth-first traversal of the permutations tree. Clearly, if the search visits the whole tree, any possible mapping would be found. On the other hand, ECF does not visit the whole tree, but prunes a branch as soon as it finds that there are no feasible mappings for the current query node, at which point any child branches also fail to satisfy the constraints for the current node and therefore it does not drop any feasible solutions.

Worst Case Performance of ECF: In the worst case, the constraints are loose enough so that any node in the hosting network is a valid candidate for each query node, and any edge as well. The hosting

[4] When the network is represented by a directed graph

```
function beginSearch()
root ← createRoot
Candidates ← set of real nodes defined by (1)
call search(root, Candidates)

function search(node, Candidates)
if node is at depth N_Q
        report mapping defined by branch from node to root
        return
for each c in Candidates
        add c as a child of node
        NextCandidates ← set from filter (2)
        if NextCandidates is empty return
        call search(c, NextCandidates)
        remove child c from node
```

Fig. 4. The Exhaustive Search with Constraint Filtering (ECF) Algorithm.

infrastructure in the worst case is a clique (any graph of $N_Q < N_R$ nodes is a subgraph of a clique of size N_R). Under these circumstances both heuristics fail to prune any candidates and ECF will explore the entire permutations tree. In Section VII we present an experimental evaluation under more realistic application conditions.

B. Random Walk Search with Backtracking (RWB)

ECF performs an exhaustive search and retrieves *all* feasible mappings for a given query. However, for many applications, finding *all* feasible embeddings is not necessary. For example an application may just require the identification of *any* (single) feasible embedding, or it may require the exploration of a representative subset of feasible embeddings (*i.e.*, a region of the solution space) in order to optimize resource allocation over that subset. For such applications, or in general when obtaining the entire set of candidate embeddings is not necessary, and/or if the problem size is too large to be handled by an exhaustive method, a non-deterministic version of ECF may be more appropriate. We describe a variant of the ECF algorithm along these lines below.

The algorithm uses the same filtering conditions (1) and (2) to randomly choose the next mapping. If at some point the algorithm reaches a dead end, it backtracks to the previous virtual node and selects another candidate. If at some level there are no candidates left, it backtracks to the parent node, and so on. If by backtracking the algorithm reaches the root node, it returns with no solution. By virtue of the randomness with which candidate mappings are selected, and the backtracking-nature of the search for a feasible embedding, we refer to this algorithm, shown in figure 5, by the Random Walk with Backtracking (RWB) algorithm.

C. Lazy Neighborhood Search (LNS)

Both the ECF and RWB algorithms have a potentially problematic attribute: their space requirements may become prohibitive, especially for under-constrained queries. In particular, the worst-case space requirement for the two filter matrices used in both ECF and RWB is $O(n|E_Q||E_R|)$. This case could occur for under-constrained queries over dense networks, since most links would be a match of each other and each filter position could map close to n candidates. Roughly speaking, the query size could also be close to the hosting network, so for dense graphs $|E_Q| \sim |E_R| \sim n^2$, so the worst case space is $O(n^5)$,

```
function randomWalkSearch()
current ← first virtual node
while(current is not null)
    Candidates ← set of real nodes from filter (1) or (2)
        minus discarded for current
    selected ← a random element from Candidates
    do until added a child or there are no more candidates
        if selected is compatible with current mapping
            add selected as a child of current
            current ← selected
            if current is the last node,
             return the mapping given by the
             branch from current to the root
        otherwise
            add selected to the list of discarded for current
    if could not find compatible child,
        current ← parent(current) // Backtrack
return "no solution"
```

Fig. 5. The Random Walk with Backtracking (RWB) Algorithm.

which for even small overlay, grid, and testbed networks (say a few dozen nodes) may be too large to handle as it will overrun the system's memory quickly.[5]

To reduce the space requirement, the algorithm we present in this section seeks to minimize the amount of state information kept during the search, and to making pruning decisions along the way. The main idea behind this algorithm is illustrated in Figure 6.

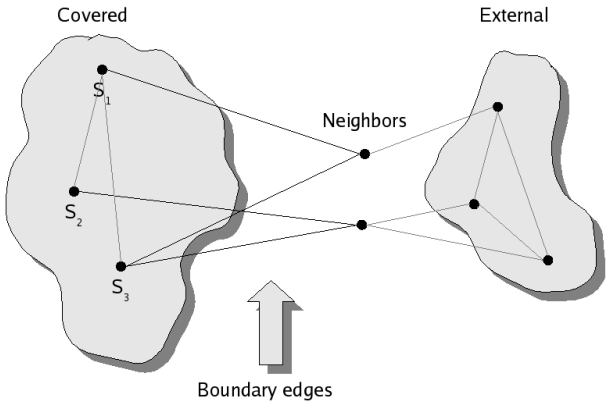


Fig. 6. Covered, Neighbor and External sets used by Lazy Neighborhood Search algorithm

At any point in time there are three sets: $Covered$, which contains the set of virtual nodes already matched, $Neighbors$, which contains the set of nodes connected to at least one covered node, and $External$, which contains the set of query nodes with no connections to the covered set. At each stage, the algorithm picks one of the neighbor vertices, and checks if there is a mapping that would allow it to satisfy topological and query constraints against all covered nodes, and if so, it adds that vertex to the

[5] Notice that if the hosting network is dense (as with overlays, in which there is an overlay link between every two nodes), then the topological constraints implied by the virtual network do not help much in reducing the number of candidate edges. This only gets worse if the query is under-constrained. In the worst case the graph could be a clique and the set of candidates would be the set of all edges going to the nodes not yet selected. In this situation the heuristic introduced in Section A of sorting by number of candidates becomes ineffective and we are truly exploring the full permutations tree.

covered set. This guarantees that all the covered vertices constitute a valid partial match. Clearly, when all query vertices are in the covered set we have a complete matching. Figure 7 shows the pseudocode of this algorithm.

Our implementation of this algorithm uses two heuristics to help prune invalid options as soon as possible: (1) In step 3, we always pick the *largest* degree virtual vertex, so that the $Covered$ set grows quickly to a set of highly connected nodes. This would ensure that neighbors will be more likely to having many links to the $Covered$ set and therefore less chances of having many valid mappings. (2) In step 5, while picking any neighbor would be correct, choosing the one with more links to the $Covered$ set forces the largest possible *conjunction* of constraints that must be satisfied, which helps prune invalid paths as soon as possible. Due to its focus on exploring neighboring nodes and its preference for nodes that would result in less mappings to check for feasibility, we refer to this algorithm as the Lazy Neighborhood Search (LNS) algorithm.

```
function LazyNeighborhoodSearch()
1)   Covered, Neighbors ← φ
2)   External sets ← set of all vertexes
3)   Pick one vertex, move it to the Covered set
4)   Vertexes connected to this move to the Neighbor set
5)   current ← neighbor vertex
6)   ConnectingEdges ← edges connecting current to covered vertexes
7)   For all possible mappings for ConnectingEdges
8)        If this mapping satisfies all constraints
9)             Add current to Covered
10)            Update Neighbors
11)            If there are no more neighbors
12)                 This is a good mapping
13)                 Return to try alternative mappings
14)            Otherwise
15)            Go recursively to step 5
16)       Otherwise try another mapping for ConnetingEdges
          and if none passes, then return there is no mapping
```

Fig. 7. The Lazy Neighborhood Search (LNS) Algorithm.

Our implementation of LNS uses a data structure that holds in parallel the sets of covered, neighbor, and external vertices from both the query and hosting networks, as well as boundary edges. It also keeps these sets synchronized when a virtual→real node-pair moves in/out of the $Covered$ set. The use of this data structure helps in several ways: (1) It facilitates the choice of the next neighbor (step 5) that has the largest number of links to the $Covered$ set. If $Neighbors$ is small, this selection will be more efficient. (2) It simplifies the identification of edges that connect the chosen neighbor to the $Covered$ nodes by keeping the set of boundary edges.[6] (3) It makes it easy to identify that a complete match has been found. Once the virtual neighbor set is empty, we know all the virtual nodes have been mapped and all the constraints are satisfied.

Correctness and Completeness of LNS: A proof of the correctness and completeness of the LNS slgorithm is given in the Appendix.

[6] Notice that this part is tricky since the correspondence of virtual boundary edges to real boundary edges is not one-to-one. This is why step 7 checks all possible mappings from a virtual neighbor to its real neighbors. For any positive match the algorithm recurses until all virtual nodes have been mapped, or no mapping is possible.

VI. NETWORK AND CONSTRAINT REPRESENTATION IN NETEMBED

To be practical, the network embedding algorithms presented in the previous section must allow for an easy interface with users and applications. For instance, such an interface should allow for an expressive specification of the topological as well as the qualitative characteristics of the hosting (real) and query (virtual) networks, and should also allow for an expressive specification of constraints on acceptable embeddings. In this section, we overview the choices we made along these lines in our NETEMBED service implementation.

A. Network Representation

Clearly, a network embedding service must adopt a "standard" representation of the various networks (graphs) that it handles, *e.g.*, query and hosting networks. To that end, there are a number of network topology representations available, but for the most part they have been designed for a specific application domain. For instance topology generators like BRITE [18] or GT-ITM [19] feature their own, different network description language. Simulation packages like ns-2 [20] are integrated with a programming language (TCL for ns-2) that provides the means to describe the network as well as to control the simulation. Experimental datasets also define their own means of characterizing the network. For example, in the particular case of the all-pairs delay traces for PlanetLab [21], the network is represented using an adjacency matrix that provides the minimum, average, and maximum delay measurements.

As varied as they are, the above-mentioned network representation approaches do not provide the generality necessary to describe the real and virtual networks with arbitrary parameters for nodes and links, as we envision for NETEMBED. For this reason we have adopted the GraphML [22] standard as a more general way to describe the networks. In GraphML a network is represented as an XML document according to the rules of the corresponding *Schema Definition*, where the top-level element is the `graph` and its children are the `node` and `edge` elements. The advantage of this representation is that is simple and flexible. Moreover, integrity (syntax and type safety) come for free with the appropriate XML parser.

A particularly attractive feature of GraphML (as it relates to the applications we envision for NETEMBED) is its support of arbitrary *typed* attributes for nodes and links. For example a topology generation tool may give us the average delay, while a measurement dataset may contain maximum and minimum delays. GraphML allows us to capture all these attributes in the network description, and by means of the constraint expression language (which we discuss next) the experimenter may establish the necessary relations between the corresponding parameters of the virtual and hosting networks.

B. Constraint Expression Language

The specification of a constraint expression (in addition to the topological constraints imposed on the links and nodes of the virtual network) allows for an independent input to our NETEMBED service and associated mapping algorithms, which enable users/applications the flexibility of specifying constraints that are separate from the query topology specification. This way adjustments can be easily made without modifying the virtual network description. This may be useful in an interactive application scenario where the user may wish to begin with more stringent constraints and relax them if there is no compliant mapping.

To provide a general framework for specifying such relationships our NETEMBED implementation includes a constraint expression language, that basically follows the rules of Java for creating boolean expressions. The language provides the standard boolean operators (&&, ||, !), relational operators (==,

Hosting Network	Virtual Network	Object
rEdge	vEdge	Edge object
rSource	vSource	Source node
rTarget	vTarget	Target node

TABLE I
OBJECTS AVAILABLE IN NETEMBED CONSTRAINT EXPRESSIONS.

$!=$, $>$, $<$, $>=$, $<=$), a basic set of arithmetic operators ($+$,$-$,$*$,$/$) and a few functions (abs, $sqrt$). It also follows the standard Java precedence rules.

The constraint expression is evaluated when comparing every edge of the virtual network with every edge of the hosting network. If such an evaluation returns a true value, the mapping between these edges is accepted. During each evaluation, the attributes of the links and nodes from both networks are available in the standard *dot notation* under the names indicated in Table I.

For example, consider a query posed against an overlay network such as PlanetLab, for which an all-pairs overlay link delay characterization is available. The virtual network may specify some requested delay values on its links, which must be matched up with overlay link delays. Furthermore, the query may specify some tolerated deviation around the requested link delays in the query network – *e.g.*, a 10% deviation around the requested delays is tolerable. The fragment below shows how such a constraint may be spelled out using our constraint expression language.

```
vEdge.avgDelay>=0.90*rEdge.avgDelay && vEdge.avgDelay<=1.10*rEdge.avgDelay
```

As another example, the fragment below specifies that a match is acceptable as long as the specified query link delay is within the minimum and maximum overlay network link delays.

```
vEdge.avgDelay>=rEdge.minDelay && vEdge.avgDelay<=rEdge.maxDelay
```

In some applications it may be necessary for some nodes in the query network to have special attributes that are not necessarily required of other nodes. To facilitate the expression of this relationship we have also provided the function `isBoundTo`. For example, if some query nodes have the attribute `osType` equals to `linux`, the following expression forces this nodes to be mapped to real nodes with the same attribute.

```
isBoundTo(vSource.osType, rSource.osType)
```

As another use example of the `isBoundTo` function, consider the case in which we need to force a particular binding. For example, assume that a given query node must have access to special hardware (*e.g.*, a particular sensor), then the query may use the attribute `bindTo` to indicate this requirement, as shown in the fragment below.

```
isBoundTo(vSource.bindTo, rSource.name)
```

As another example of how the constraint expression language could be used to relate the specifications of a virtual network to the characteristics of the hosting network, the fragment below specifies the requirement that, in any valid embedding, the geographic distance between the desirable location of a query node and its corresponding hosting network node cannot exceed (say) 100 meters.

```
sqrt( (vSource.x-vTarget.x)*(vSource.x-vTarget.x) +
      (vSource.y-vTarget.y)*(vSource.y-vTarget.y) ) < 100.0
```

Our NETEMBED implementation uses the well known tools JFlex[23] and CUP[24] to implement the lexer and parser of the expression language.

VII. PERFORMANCE EVALUATION

In this section, we present results of network embedding experiments we have conducted using NETEMBED.

A. Experimental Setting

For each one of our experiments, and as illustrated in Figure 1, two "networks" need to be provided to NETEMBED: the query (or virtual) network, and the hosting (or real) network.

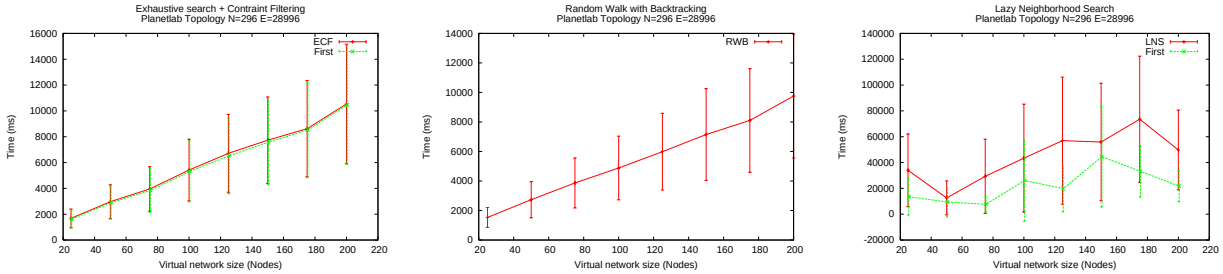
In the presentation that follows, our hosting networks were either real infrastructures, namely PlanetLab [11], or else they were generated synthetically using Internet topology generators, namely BRITE [18].

The generation of the networks used as queries is a bit trickier. In the presentation that follows we have adopted one of three approaches in generating the query networks. Using the first approach, the query network is a (typically small) subgraph selected at random from the hosting network, *e.g.*, PlanetLab. Such queries would be typical for applications that require the instantiation of Internet-like topologies, for example, as would be the case for configurations requested by a user of PlanetLab. One advantage of using this approach is that since the query is "sampled" from the hosting network, we know that an embedding exists. This provides us with good "test cases" for NETEMBED. Using the second approach, query networks are regular topologies that are synthetically generated (*e.g.*, rings, stars, cliques, *etc.*). Such queries would be typical for applications that exhibit a regular communication structure, as would be the case in high-performance grid applications, for example. Using the third approach, query networks are irregular topologies that are synthetically generated using a topology generator such as BRITE. The motivation behind using this approach is similar to that of the first approach (since topology generators such as BRITE aim to generate "Internet-like" topologies). Additionally, using a synthetic generator allows us to study the effect of the topological characteristics of the query network on NETEMBED's performance, since we can control such characteristics.

The main performance metric we consider in our experiments is *response time*, which is the time (measured in milliseconds) it takes NETEMBED to answer a query. The response times reported for all experiments presented in this section were obtained by running NETEMBED on an Intel Xeon 2Ghz system with 1GB of main memory (enough for NETEMBED to avoid any noticeable paging).

B. Evaluation Using Planetlab

In these experiments we used the PlanetLab all-pairs ping trace [21] to generate the hosting network. This data set provides maximum, minimum and average delay between PlanetLab sites. Although there are 296 sites in the trace, some of the sites might not have been running the deamon or were down, so the actual number of active sites is a little lower and the underlying graph is not a clique. In any case, the network has 28,996 edges, providing a rich and large enough network for our tests.



(a) ECF (b) RWB (c) LNS

Fig. 8. Mean Search Time for Planetlab subgraphs using the ECF, RWB, and LNS algorithms.

The query networks for the queries were generated as random connected subgraphs from the hosting network (obtained as described above) of size N nodes. As we alluded before, setting the query to be a subgraph from the hosting network guarantees there is always at least one match. For each size query network size, queries were produced by varying the number of edges (E), and for each (N,E)-pair we had 5 different queries, so the results were not biased by a particular network configuration. The network embedding algorithms were run for each different query using the same constraint expression in all cases, namely that the real link delay range is within the specified query-link delay range.

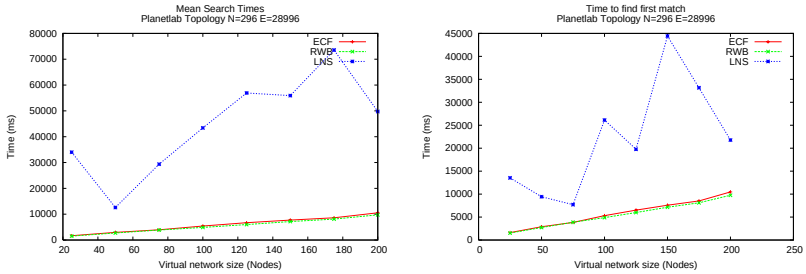
Figure 8 (a) shows the performance results for the ECF algorithm. For each data point, the average and the confidence interval are shown. A second line indicates the time to find the first match, which is an interesting performance measure for applications that require just a single feasible embedding. Given the fixed size of the hosting network, we limit the queries to be up to 200 nodes, and for the largest cases we had running times around 14 seconds on a Xeon 2.5Ghz system. It is worth noticing that, having a fixed-size hosting network, the search times seem to grow linearly with the size of the query, indicating that our filtering heuristic has been quite effective in avoiding the complexity associated with the full exploration of the search space (which grows quadratically with the size of the query network).

An interesting observation from Figure 8 (a) is that the difference between the time to retrieve *all* matches and the time to find the *first* match is very small, indicating that most of the time was spent in the unmatched region of the search space – *i.e.*, once solutions are found, many similar solutions are found by varying just a few nodes (close to the leaves of the tree).

Figure 8 (b) shows the time to find the *first* solution using the RWB algorithm. In this case we also observe that the mean search time shows a very linear relationship with the size of the query network.

Finally, the results for the LNS algorithm are shown in Figure 8 (c). Interestingly, LNS requires essentially constant time to find matches *independent* of the query network size. Even more interesting is the fact that for large query networks there is a slight tendency for the time to decrease. This behavior could be explained by noting that in large networks, there are usually very few matches, and growing a partial match implies matching a larger number of constraints.

Figure 9 provides a comparison of the three algorithms. Figure 9 (a) shows the mean time until all matches are found, whereas Figure 9 (b) shows the time until the first match is found. The ECF and RWB algorithms have very similar performance, making clear that the filtering strategy to prune the permutations tree is the prime contributor to the performance of these two algorithms, and that once a solution is found, most of the other solutions are close-by. On the other hand, it is interesting to observe that on average the LNS algorithm is very slow in comparison, but when considering only the first match, its performance is not too far off. This could be explained by noting that LNS does not have the advantage of sorting the



(a) (b)

Fig. 9. Comparison of the three algorithms running queries on Planetlab

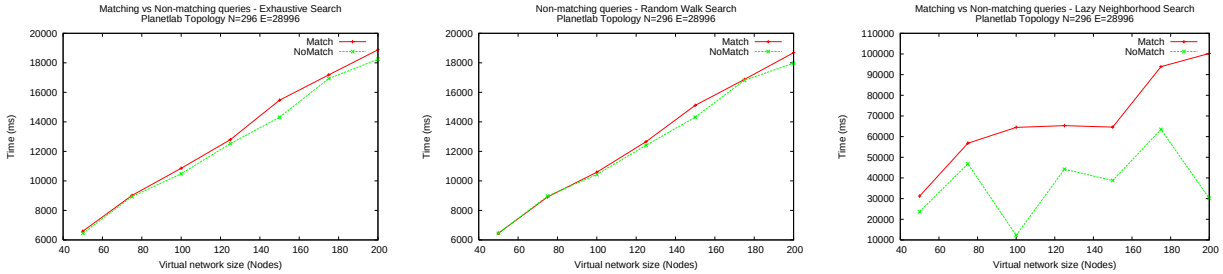


Fig. 10. Search times for feasible versus infeasible queries on PlanetLab topologies

edges by number of potential matchings. Thus, when an edge that produces many matches appears early in the selection process, it will result in the repetition of a lot of redundant work for all the subsequent matches, which will result in significantly larger times if all matches are to be produced.

Another important question regarding the performance of the various algorithms is the time it takes them to conclude that an embedding is not feasible. To evaluate this, we performed two sets of experiments using PlanetLab as the hosting network. The first set of experiments used queries that were known to be feasible (as done above). The second set of experiments used queries that were known to be infeasible. The infeasible queries were generated from the feasible queries by changing some of their link attributes (*e.g.*, delays) to some infeasible values. Notice that doing so does not change the topology of the query network, only the constraints imposed on what would constitute a feasible embedding. Figure 10 shows the results for the three algorithms. In general, the performance for ECF and RWB is very similar which is congruent with the fact that these algorithms will explore a significant portion of the search tree in either case. LNS is noticeably slower. However, it determines the non-existance of feasible matches (*no-match* results) in less time.

C. Evaluation Using Synthetic BRITE Topologies

The fixed size of PlanetLab limits us from exploring the scalability of our algorithms on larger hosting networks. Also, since the delays on PlanetLab links are characterized using end-to-end measurements (pings), the resulting topology is mostly a clique, which is far from being representative of physical (as opposed to overlay) Internet topologies. For these reasons, and as we alluded earlier, we also considered synthetic topologies generated using the BRITE topology generator, based on the power-law models of node connectivity of the Internet.

For these experiments we generated 3 hosting networks with 1500, 2000 and 2500 nodes respectively. For each one of these networks, we obtained various sets of subnetworks of different sizes. The results

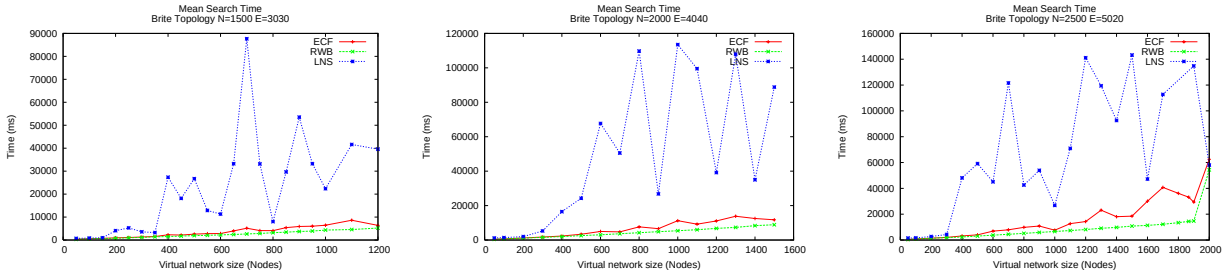


Fig. 11. Mean Search time for BRITE topologies

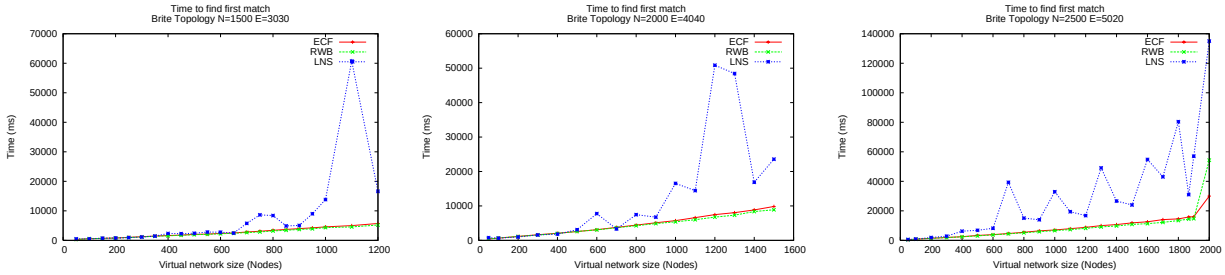


Fig. 12. Mean Time to Find First Match in BRITE topologies

of the three algorithms for these networks are shown in Figure 11. In general, and not surprisingly, the behavior follows the same pattern observed with PlanetLab topologies: Similar average times for ECF and RWB, with a slight advantage to the latter. High variability and much larger mean times with LNS. Figure 12 shows the mean time to find the first match for these BRITE cases. Again, the difference between the first two algorithms and LNS is not so pronounced.

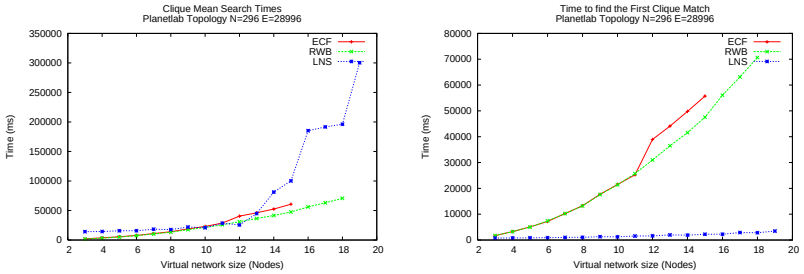
D. Evaluation Using Queries with Regular Topologies

The two characteristics that make an embedding difficult to find are: (1) under-constrained queries, and (2) queries with regular topologies. Under-constrained queries do not provide enough conditions to significantly prune the search space. In the limit, the only constraint is that of the query topology, and the problem is reduced to a subgraph isomorphism problem. With regular topologies (such as cliques, rings, stars with equal or no constraints on all edges), any permutation of a partial match is also a partial match. Thus, if this partial match leads to a dead end, the embedding algorithms will end up performing the same amount of (useless) work on every permutation.

To evaluate the performance of our algorithms under these "worst-case" scenarios, we used as queries a series of cliques of increasing size, whose only constraint was to have a end-to-end delay between 10 and 100ms. We then try to find matches on PlanetLab for each one of these cliques. The query is under-constrained as there are about 6,700 edges that fall in these delay ranges and the query topology is regular.

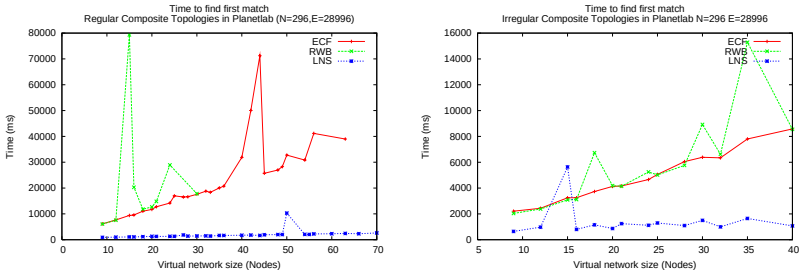
Figure 13 (a) shows the mean time to find *all* embeddings of a clique in PlanetLab as a function of the clique size. Those cases in which no solutions were found, or in which the algorithm timed-out before returning any solution are excluded, so as not to affect the trend of the graph. It is worth noting that under this query model, LNS always times out before it is able to find *all* embeddings.

Figure 13 (b) compares the three algorithms using the time to find the first match. In this case, the LNS algorithm greatly outperforms the other two. When it finds a solution it finds it quickly as the heuristic to grow the matching with the vertex with more constraints helps prune non-matching cases rapidly as this



(a) (b)

Fig. 13. Finding Matchings for a Clique in Planetlab



(a) Regular link constraints (b) Irregular link constraints

Fig. 14. Finding matchings for composites of regular topologies

forces each new vertex to match all the already selected vertexes. On the other hand, the regular structure forces the algorithm to start over when finished with a candidate matching explaining the long running times for the average case.

Our last set of experiments we considered involved composite queries. A composite query is a two-level hierarchical topology, where both levels have regular structures. So for example the root level could be a ring, a star, or a clique, and each vertex of the root level is also a regular structure. Many practical applications follow these kinds of structures, including multicast trees, distributed hash tables, and rings, to name a few. We prepared two sets of queries. The first set has regular constraints per level – namely, all links at the root level have a delay constraint between 75 and 350ms, representing inter-site wide-area delays, and all the links on the lower level have delay constraints between 1 and 75ms, representing intra-site delays. These values of the delays were chosen from the distribution of delays in PlanetLab so that there are abundant links in both ranges. The second set shares the same topological structures as with the first set, but features delay constraints that are randomly assigned from the 25-175ms range, which contains about 70% of the links in PlanetLab. For both sets, there are usually thousands of matchings for each query, so the interesting measure here is the average time to find the first match. Figure 14 shows the results for (a) the regular and (b) the random constraint assignment cases.

The interesting observation about these two cases is that (as with the first match in the case of cliques) LNS finds the first solution in almost constant time and by far outperforms the other two algorithms. This reinforces the previously mentioned hypothesis that in under-constrained queries and high-density graphs LNS is better suited to find the first solution.

E. Quality of Returned Results

Using any one of our embedding algorithms, NETEMBED may return one of three types of results: (1) The complete set of all feasible embeddings, (2) A subset of all feasible embeddings, and (3) An inconclusive

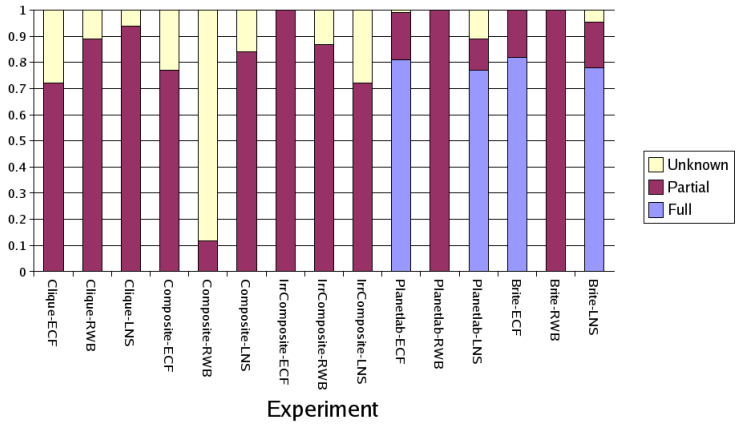


Fig. 15. Probability distribution of the different types of results.

response.

The complete set of all feasible embeddings (including none, if the query network is impossible to embed) is returned when the algorithm terminates before its preset timeout has expired. A partial set (subset) of all feasible embeddings is returned when the algorithm times out after finding some (but not necessarily all) feasible embeddings.[7] Finally, the algorithm is said to have an inconclusive response, if if fails to produce any feasible embedding by the timeout. Notice that in this case, it is inconclusive whether or not a single feasible embedding exists.

Figure 15 shows the probability of each one of these results for all the experiments presented earlier in this section. Except for one case, in which RWB behaves poorly, the probability of finding matches was over 70%. Even more impressive, for some types of queries, ECF and LNS were able to find all feasible matches with a probability of 75% to 82%. Looking at the probability of finding any embedding (as opposed to all embeddings) for ECF and LNS, we observe that for queries with regular topologies (clique and composite), LNS has a better chance of success. This, combined with the much better performance in terms of response time, makes LNS ideal for these kinds of queries. On the other hand, for very constrained queries, where filtering results in much more effective pruning, ECF outperforms LNS in both chances of success and response time.

F. Performance Results Relative to Existing Techniques

Previously published techniques ([13], [10], [14], [15], [16], [17]) give us a baseline for the purposes of comparison. Most of these existing techniques handle only small networks, typically tens of nodes, and have reported response times in the order of tens of minutes or more. We have shown that depending on the query type, our algorithms handle much larger networks and have response times that go from as low as fractions of a second to a few minutes in the worst case.[8]

[7] Notice that RWB will always return a partial result (or no results) since by design it terminates as soon as it finds the *first* solution.

[8] It is hard to make a direct comparison between these various techniques since, as we alluded in Section II, most of these techniques are designed to find optimal mappings and/or are designed to deal with very particular kinds of mappings. That said, we emphasize that optimality conditions can be integrated in any of our algorithms by using it in candidate selection and/or in the pruning strategy.

VIII. CONCLUSION

Summary: We presented three new algorithms for the solution of the network embedding problem. Our algorithms were shown to outperform previously proposed approaches in terms of their scalability and response time for practically-sized problems (*e.g.*, finding mappings in a testbed network as large as Planetlab, or in Internet-like networks of reasonable size). Our methods leverage a number of effective heuristics that reduce the search space. Correctness and completeness are ensured by exploring the search space in an ordered fashion. We have incorporated our network embedding algorithms in NETEMBED – a service that allows applications to identify feasible embeddings of requested virtual topologies into a hosting network. NETEMBED features a general constraint specification framework as well as a standard network representation using GraphML.

The experimental evaluation of NETEMBED we presented in this paper revealed an almost linear scalability with respect to the query (virtual network) size, when the hosting infrastructure is fixed in size. We have shown that two of our algorithms (ECF and RWB) perform well in situations where the query is tightly constrained, and when the network density is low. On the other hand, our third algorithm (LNS) performs much better with less constrained queries and higher density networks.

Current and Future Work: The work presented in this paper is one component of a framework for allocating resources in a distributed network: Finding the resources that meet the given requirements. The mapping of the needed resources may not exist, may be unique, or there may be possibly many satisfactory mappings. In the former case, an interesting follow-up work, which we are pursuing, is to allow many-to-one mappings between virtual and real nodes (*e.g.*, by mapping a link in the query network to a path in the real network). In the latter case, an interesting follow-up work is to consider the optimization problem: What assignment of resources minimizes some cost metric or objective function?

Another important problem is considering temporal relationships, *i.e.*, the scheduling problem. When used in a real application, resources once assigned would not be available for some amount of time. In such settings, the embedding problem must be tightly integrated with the scheduling problem – to find a window of time (or the closest window of time) in which some feasible embedding is available. In addition, this may be combined with the optimization problem of finding an optimal schedule when multiple embeddings are available. We are pursuing this within the context of the SNBENCH sensor networks framework developed at Boston University [25]. SNBENCH targets complex environments where applications may require specific allocations of various sensing, computing, and communication resource capacities. Since multiple applications may share this infrastructure, such an environment requires an integrated mapping and scheduling service to coordinate the sharing of resources between applications.

As described, our NETEMBED service is a "centralized" service, in the sense that it has a complete view of both the hosting and query networks. For truly large-scale networks, a complete view of the network may not be available to a single domain (or authority). Thus, it is desirable in such settings for services such as NETEMBED to be implemented in a distributed fashion, which would be advantageous for both service scalability and management. We are currently looking into a hierarchical approach to a decentralized implementation of NETEMBED.

IX. PROJECT WEB SITE

Additional information as well as the NETEMBED software is available at: `http://csr.bu.edu/netembed`

APPENDIX

Reducing the Size of the Permutations Tree for ECF and RWB:

Lemma 1 *(Minimum Size of the Permutations Tree)*
If the virtual nodes are sorted by increasing number of candidate mappings, then the total number of nodes in the permutations tree is minimum.

Proof: Let n_i be the number of candidate mappings for virtual node v_i. According to the statement in the Lemma, nodes are examined in increasing order of their candidate mappings. Thus, $n_i < n_{i+1}$ for $1 \leq i < N_Q$ and the total number of nodes in the permutations tree is given by

$$S = n_1 + n_1 n_2 + \ldots + n_1 n_2 \cdot \ldots \cdot n_{N_Q} = n_1(1 + n_2(1 + \ldots n_{N_Q-1}(1 + n_{N_Q}) \ldots)) \tag{3}$$

Assume that there is an alternative ordering of nodes that yields a smaller total number of nodes (namely S') in the permutations tree—*i.e.*, $S' < S$. In particular, without loss of generality, assume that such an ordering would switch the order with which we examine nodes v_1 and v_2, which yields the following

$$S' = n_2(1 + n_1(1 + \ldots n_{N_Q-1}(1 + n_{N_Q}))) \tag{4}$$

Since, by definition, $n_2 < n_1$, we get

$$S' - S = n_2 - n_1 > 0 \tag{5}$$

This implies that the alternative ordering yields a larger total number of nodes—*i.e.*, $S' > S$—which is a contradiction. Thus, by having n_1 as the first factor minimizes the total number of nodes if we keep fixed the order of the rest of the nodes. The same procedure can also be applied to the successive factors $(1 + \ldots n_{N_Q-1}(1 + n_{N_Q}))$ so that each consecutive internal factor is minimized as well. This allows us to conclude that ordering the mappings in increasing order by the number of candidates minimizes (3). ■

Correctness and completeness of LNS:

Definition *1:* A *promising mapping* is a subset of any complete mapping. Observe that, if there is a mapping, the empty set is always a promising mapping.

Lemma *2:* If, at the current state, the covered set of the LNS algorithm is a promising mapping of size k, then LNS will find a promising mapping of size $k + 1$

Proof: Let P_k be a promising mapping of size k at some stage of the LNS algorithm. Then, the next neighbor selection heuristic selects a neighbor node n and tries all possible mappings r for this node. As P_k is promising, then there must be at least one valid mapping for n and by trying them all, LNS will find it and continue with the promising set of size P_{k+1}. ■

Theorem *1:* If there is any feasible mapping, the LNS algorithm will find it. If not, the algorithm will exit with empty solution.

Proof: The proof of the first part of the theorem is evident by induction. The algorithm starts with the empty set which is promising. By lemma 1 it will find any promising mapping of size 1. At any state k it will find the promising mapping of size $k+1$ and when $k+1 = N_Q$ these are complete mappings. If there is no solution, the algorithm tries all extensions of the empty mapping, and after finding none, returns the empty set. ■

The completeness of the LNS algorithm follows from the fact that at any state P_k, LNS tries all possible mappings for the next neighbor and therefore it will find all feasible promising sets P_{k+1}, if any.

REFERENCES

[1] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, , and John Kubiatowicz, "Maintenance-free global data storage," *IEEE Internet Computing*, vol. 5, no. 5, pp. 40–49, September/October 2001.

[2] Emil Sit, Frank Dabek, and James Robertson, "UsenetDHT: A low overhead usenet server," in *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, San Digeo, CA, February 2004.

[3] Suman Banerjee, Seungjoon Lee, Bobby Bhattacharjee, and Aravind Srinivasan, "Resilient multicast using overlays," in *Proc. of ACM Sigmetrics*, June 2003.

[4] J. W. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," in *Proc. of ACM SIGCOMM*, Pittsburgh, PA, August 2002, pp. 47–60.

[5] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh, "Splitstream : High-bandwidth content distribution in a cooperative environment," in *Proceedings of 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[6] Y.-H. Chu, S. Rao, S. Seshan, and H. Zhang, "A case for end system multicast," *IEEE Journal on Selected Areas in Communications*, vol. 20(8), pp. 1456–71, October 2002.

[7] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat, "Bullet: high bandwidth data dissemination using an overlay mesh," in *Proc. of ACM SOSP*. 2003, pp. 282–297, ACM Press.

[8] Gu-In Kwon and John W. Byers, "Roma: Reliable overlay multicast with loosely coupled tcp connections," in *Proceedings of IEEE Infocom '04*, Hong Kong, March 2004.

[9] Virginia Lo, Daniel Zappala, Dayi Zhou, Yuhong Liu, and Shanyu Zhao, "Cluster computing on the fly: P2P scheduling of idle cycles in the Internet," in *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, San Digeo, CA, February 2004.

[10] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002, pp. 255–270.

[11] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe, "A blueprint for introducing disruptive technology into the internet," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 59–64, 2003.

[12] "The GENI initiative," http://www.nsf.gov/cise/geni/.

[13] Chris Alfeld, Jay Lepreau, and Robert Ricci, "A solver for the network testbed mapping problem," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 65–81, 2003.

[14] B. White, J. Lepreau, and S. Guruprasad, "Lowering the barrier to wireless and mobile experimentation," in *In Proceedings of Hotnets-I*, Princeton, NJ, October 2002.

[15] Yong Zhu and Mostafa Ammar, "Algorithms for assigning substrate network resources to virtual network components," in *INFOCOM*, April 2006.

[16] Jeffrey Considine, John W. Byers, and Ketan Mayer-patel, "A constraint satisfaction approach to testbed embedding services," in *Proceedings of HotNets-II*, November 2003.

[17] David Oppenheimer, Jeannie Albrecht, David Patterson, and Amin Vahdat, "Design and implementation tradeoffs for wide-area resource discovery," in *14th IEEE Symposium on High Performance Distributed Computing (HPDC-14)*, July 2005.

[18] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers, "BRITE: Universal topology generation from a user's perspective," Tech. Rep. 2001-003, 1 2001.

[19] Ellen W. Zegura, Ken Calvert, and S. Bhattacharjee, "How to model an internetwork," in *Proceedings of IEEE Infocom '96*, IEEE, Ed., 1996.
[20] "The network simulator, ns-2," http://www.isi.edu/nsnam/ns/.
[21] Chad Yoshikawa, "All-sites-pings for planetlab," http://ping.ececs.uc.edu/ping/.
[22] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M.S. Marshall, "Graphml progress report: Structural layer proposal," in *Proc. 9th Intl. Symp. Graph Drawing (GD '01), LNCS 2265*, Springer-Verlag, Ed., 2001, pp. 501–512.
[23] "Jflex," http://www.jflex.de/.
[24] "Cup," http://www2.cs.tum.edu/projects/cup/.
[25] Michael Ocean, Azer Bestavros, and Assaf Kfoury, "snBench: Programming and virtualization framework for distributed multitasking sensor networks," in *Proceedings of the 2nd international conference on Virtual execution environments (VEE 2006)*, New York, NY, USA, June 2006, pp. 89 – 99, ACM Press.
[26] Robert Ricci, David Oppenheimer, Jay Lepreau, and Amin Vahdat, "Lessons from resource allocators for large-scale multiuser testbeds," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 1, pp. 25–32, 2006.
[27] Matthew L. Massie, Brent N. Chun, and David E. Culler, "The ganglia distributed monitoring system: Design, implementation and experience," .
[28] KyoungSoo Park and Vivek S. Pai, "Comon: a mostly-scalable monitoring system for planetlab," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 1, pp. 65–74, 2006.
[29] KyoungSoo Park and Vivek Pai, "Cotop: A slice-based top for planetlab," http://codeen.cs.princeton.edu/cotop/.
[30] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris, "Vivaldi: a decentralized network coordinate system," pp. 15–26, 2004.
[31] Paul Brett, Rob Knauerhase, Mic Bowman, Robert Adams, Aroon Nataraj, Jeff Sedayao, and Michael Spindel, "A shared global event propagation system to enable next generation distributed services," in *WORLDS04: First Workshop on Real, Large Distributed Systems*, 2004.