

# NETEMBED: A Network Resource Mapping Service for Distributed Applications<sup>†</sup>

Jorge Londoño<sup>‡</sup>                      Azer Bestavros  
jmlon@cs.bu.edu                      best@cs.bu.edu  
Computer Science Department  
Boston University

## Abstract

*Emerging configurable infrastructures (large-scale overlays, grids, distributed testbeds, and sensor networks among others) comprise diverse sets of computing resources and network conditions. The distributed applications to be deployed on these infrastructures exhibit increasingly complex constraints and requirements on the resources they require. 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. We present two new techniques to tackle this combinatorially-hard problem that thanks to a number of heuristics, are able to find feasible solutions or determine the non-existence of a solution in most cases, where otherwise the problem would be intractable. These techniques are also false negative free, a common problem among other heuristics currently in use.*

## 1 Introduction

**Motivation:** A common problem when deploying a distributed application is that of selecting the resources to be used. It is well known that the choice of resources plays a major role in determining the application’s performance. A canonical example are overlay networks of end-systems, such as those required in grid computing environments [7, 13, 6, 5, 14].

**Challenge:** Given the requirements of an application and the characteristics of the underlying hosting infrastructure,

<sup>†</sup> This work is supported in part by a number of NSF awards, including CISE/CSR Award #0720604, ENG/EFRI Award #0735974, CISE/CNS Award #0524477, CNS/NeTS Award #0520166, CNS/ITR Award #0205294, and CISE/EIA RI Award #0202067.

<sup>‡</sup> 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”.

find (if possible) a mapping between the nodes and links of the query and those of the hosting underlying infrastructure. Since the the size of the query for any practical application is much smaller than that of the hosting infrastructure, the mapping problem is better viewed as an *embedding problem* – that of embedding the a *virtual graph* (the query) into the hosting environment.

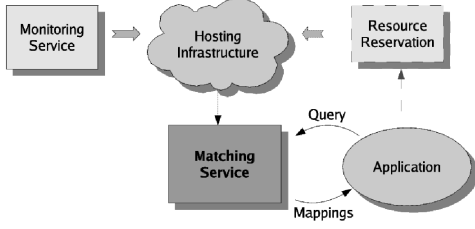
At its heart, the network embedding problem is a combinatorial search problem. In practice, however, the more constrained the virtual topology is, the less the number of links/nodes that will satisfy the constraints. Such simple observation allows us to prune significant portions of the search space, turning what would be otherwise a computationally-impractical problem into a problem solvable for queries of sizes at least as large as what is commonly found in practice.

**Contribution:** We present various heuristics to efficiently handle this problem. They do so by pruning large portions of the search space, but contrary to other well known techniques ours do not overlook any valid embeddings: If an embedding is possible, our approach will find it, if given enough time. We also present an implementation of our proposed techniques, which we call NETEMBED, together with extensive performance evaluation experiments using several combinations of synthetic queries into PlanetLab as hosting environment. Our results show that NETEMBED is quite effective in identifying one (or all) possible embeddings for quite sizable test cases, while offering a much broader scope than currently available techniques.

## 2 The netEmbed Service Model

The embedding service would include the components illustrated in Figure 1:

1. A model of the real infrastructure that characterizes the resources available. Such model could be maintained



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

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.
3. Optionally, if a resource reservation system is in place, applications would allocate the selected mapping and the network model would be adjusted accordingly.

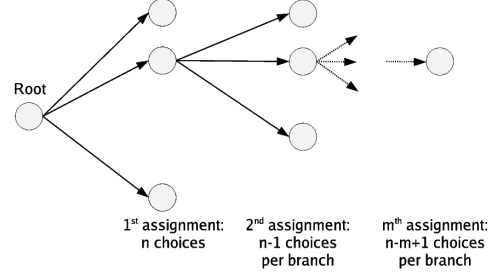
In the present work, we are focusing on the problem of finding *feasible mappings*. An interesting future extension of this work would involve the consideration of optimality functions and being able not only to find the optimal by evaluating the cost of each feasible solution, but to use the optimality condition to guide/prune the search process itself.

### 3 Basic Definitions and Terminology

We use the term *Hosting Infrastructure* to refer to the target of an embedding service. A hosting infrastructure is described as a labelled graph  $R = \langle V, E \rangle$ . Labels for nodes and edges may include numeric values or ranges and categorical classes such as “*Link (n<sub>1</sub>, n<sub>2</sub>) is Myrinet*” or “*node n<sub>1</sub> is linux-2.6*”, for example.

We use the term *Query* to refer to the application that needs to be embedded into the hosting infrastructure. A query is a labelled graph  $Q = \langle V, E \rangle$ , whose labels represent constraints given by the application.

We use the term *Mapping* to refer to a one-to-one (injective) function  $m : Q \rightarrow R$ , such that for all query nodes  $n_Q \in Q$ ,  $n_R = m(n_Q)$  is the corresponding node in the hosting infrastructure, such that all node and edge



**Figure 2. Illustration of the permutation tree that defines the search space.**

constraints are satisfied. To simplify notation, we will use  $q \rightarrow r$  to indicate that node  $q$  maps to node  $r$ .

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/edges to hosting infrastructure nodes/edges. NETEMBED defines a simple constraint expression language, which we describe later in Section 5.

## 4 netEmbed Mapping Algorithms

### 4.1 Exhaustive Search with Constraint Filtering (ECF)

This algorithm is an improved branch and bound, where the search space takes the form of a tree. As illustrated in Figure 2, the first level represents all potential mappings  $r_i$  for the first query node  $q_1$ . There are at most  $n$  such mappings. Once  $q_1$  has been assigned, there are at most  $n - 1$  choices for  $q_2$  and so for. If  $n = |N_R|$  and  $m = |N_Q|$ , the size of the search space is  $P_m^n$ , which is prohibitively expensive even for moderate values of  $n$  and  $m$ . Three strategies help expedite the search:

1. *Pruning*: If the mapping up to the current node is infeasible, the entire branch derived from this node is infeasible and it is pruned.
2. *Reordering*: If we knew that there are  $n_i$  nodes at level  $i$ , then reordering the tree so that  $n_i < n_j$  for  $i < j$

$j$  reduces the total number of nodes in the tree, thus minimizing the search space.

3. *Filtering*: By checking individual node and edge constraints we can construct a candidates filter  $F$ , which serves two purposes: use topological constraints to prune infeasible mappings and estimating the number of candidates per query node.

Thus, in the first stage of the ECF algorithm, it evaluates the constraint expression for every possible pair of virtual and real edges, producing a list of candidate mappings of the form:

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

Each cell in the candidates matrix  $F$  has coordinates  $(v, r, v_s)$ , and contains the set of candidate mappings for  $v_s$ , when  $v$  is mapped into  $r$ . Therefore, each matching adds an element to  $F$ :

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

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

The ECF algorithm then proceeds as follows:

- (1) Pick the first virtual node  $v_s$  as the one with the least number of candidates. Being the first, it can be chosen from:

$$\bigcup_{\text{all } v \in N_Q, r \in N_R} F[v, r, v_s] \quad (1)$$

- (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. This process guarantees that each additional node mapping is consistent with the topology and the established constraints.

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

Once  $N_Q$  mappings have been found using the above process, then this is a valid mapping for the whole query. The complete algorithm incorporating these filters is given in Figure 3.

Due to space limitations, we refer the reader interested in a more formal analysis to [8].

```

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

function search(node, Candidates)
  if node is at depth NQ
    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

```

Figure 3. The Exhaustive Search with Constraint Filtering (ECF) Algorithm.

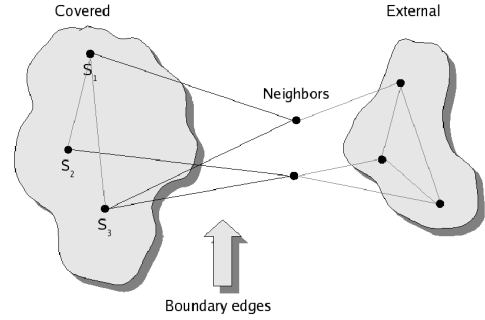


Figure 4. Covered, Neighbor and External sets used by Lazy Neighborhood Search algorithm

## 4.2 Lazy Neighborhood Search (LNS)

Worst case space requirements for ECF are  $O(n^A)$ . This turns out in practice to be prohibitive for under-constrained queries, *i.e.*, cases where the constraints do not significantly reduce the number of candidates. This motivates the development of a second algorithm which seeks to minimize the amount of state information kept during the search. The main idea behind this algorithm is illustrated in Figure 4.

At any point in time there are three sets: *Covered*, which contains the set of query 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 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 5 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 query 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 ←  $\phi$ 
2) External sets ← set of all vertices
3) Pick one vertex, move it to the Covered set
4) Vertices connected to this move to the Neighbor set
5) current ← neighbor vertex
6) ConnectingEdges ← edges connecting current to covered vertices
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

```

**Figure 5. The Lazy Neighborhood Search (LNS) Algorithm.**

Due to space limitations, we refer those interested in a more in depth analysis to [8].

## 5 Constraint Representation in netEmbed

To provide a general framework for specifying constraints between the query and the hosting infrastructure 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

(==, !=, >, <, >=, <=), a basic set of arithmetic operators (+, -, \*, /) and a few functions (*abs*, *sqrt*).

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 feasible. The attributes of the links and nodes for both, query and hosting network, are available in the standard *dot notation* as illustrated in the example below.

We also found to be a very common need to be able to bind an attribute of a query object to an attribute of the hosting object, but only for some objects. For that purpose the language also provides the function `isBoundTo`.

As a concrete 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, and requested bandwidth is less or equal to the available bandwidth. It also requires that if the virtual node has an attribute `osType`, then the matching node must have the same attribute value.

```

vEdge.avgDelay>=rEdge.minDelay &&
vEdge.avgDelay<=rEdge.maxDelay &&
vEdge.availBw<=rEdge.availBw &&
isBoundTo(vSource.osType, rSource.osType)

```

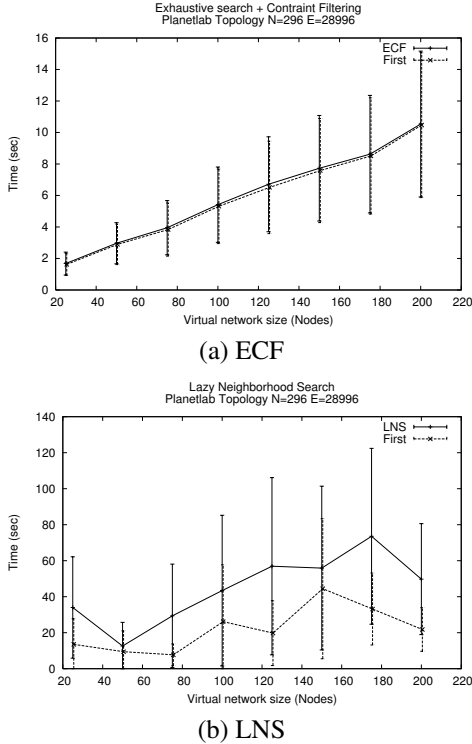
Our NETEMBED implementation uses the well known tools JFlex[2] and CUP[1] to implement the lexer and parser of the constraint expression language.

## 6 Performance Evaluation

### 6.1 Experimental Setting

For each one of our experiments we must provide two graphs to NETEMBED: the hosting infrastructure and the query. As a infrastructure for the deployment of our queries, we chose PlanetLab [11] as it provides a very representative sample of the connectivity between research centers throughout the Internet.

The generation of the queries is a bit trickier. In the presentation that follows we have adopted one of three approaches in generating the query networks. The first one is to pick a subgraph of the hosting infrastructure as the query. One advantage of using this approach is that, since the query is “sampled” from the hosting network, we know in advance that an embedding exists. This provides us with good test cases for true positives. By randomly modifying a few nodes/edges on the query, giving infeasible values to some of their attributes, we obtain our second set of



**Figure 6. Query Time using the ECF and LNS algorithms.**

queries that provides true negatives. Finally, by using synthetically generated graphs with a regular structure, such as trees, rings, stars, cliques, *etc.*, we obtain queries typical for cases such as the parse trees of a large composition of functions, or applications that exhibit a regular communication structure, as are many P2P or DHT applications.

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

## 6.2 Evaluation using random graphs

In these experiments we used the PlanetLab all-pairs ping trace [16] as model of the hosting network. This dataset provides maximum, minimum and average delay between PlanetLab sites. Notice that there are 296 sites in the trace, each site hosting a few machines. The trace contains sites that were non-responding, 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 environment for our tests.

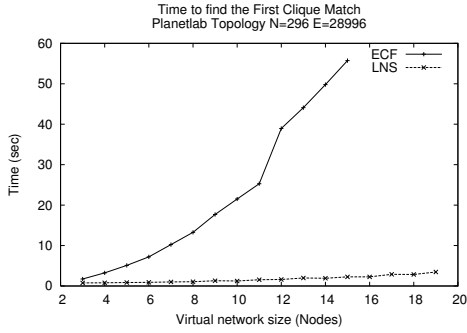
The queries were generated as random connected subgraphs from the hosting infrastructure. As we alluded before, setting the query to be a subgraph from the hosting network guarantees that there is always at least one match. For every query of size  $N$ , multiple queries were produced varying the number of edges ( $E$ ). For each  $(N,E)$ -pair we constructed 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 6 (a) shows the performance results for the ECF algorithm. For each data point, the average and the 90% 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 our 2Ghz Xeon 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.

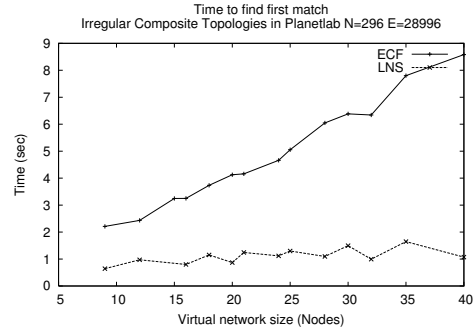
An interesting observation from Figure 6 (a) is that the difference between the time to retrieve *all* matches and the time to find the *first* match is very small, indicating that most of the time was spent computing the candidates filter and then exploring the unmatched region of the search space. Once solutions are found, many similar solutions are found by varying just a few nodes (close to the leaves of the tree).

The results for the LNS algorithm are shown in Figure 6 (b). Interestingly, LNS does not have a regular trend. We tracked the cause of the high variability to the amount of backtracking, which is dependent on the particular case and the starting condition. In general, LNS was significantly slower to explore all the feasible mappings, but if we compare the times to find the first mapping, the results are not too far away from those of the ECF algorithm.

To evaluate the case where an embedding of a random graph is not feasible we performed two sets of experiments. The first set used queries that were known to be feasible (as above) and the second set used queries that were known to be infeasible. The infeasible queries were generated from the feasible queries by changing the delay of some edges



**Figure 7. Finding Matchings for a Clique in Planetlab**



**Figure 8. Finding matchings for composites of regular topologies**

to 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. In general, the performance for ECF is very similar in both cases, indicating that after filtering, the portion of the search space explored is essentially the same. LNS is noticeably slower. However, it determines the non-existence of feasible matches (*no-match* results) in less time.

### 6.3 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 it tries, a phenomenon called *trashing*.

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 an 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 range and the query topology is regular.

Figure 7 compares the two algorithms using the time to find the first match. In this case, the LNS algorithm greatly

outperforms ECF. 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 forces each new vertex to match all the already selected vertices.

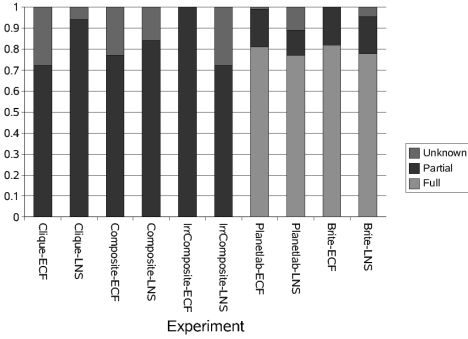
The last set of experiments consider composite queries. A composite query is a two-level hierarchical topology, where both levels have regular structures. So for example the root level could be a ring, a star, or a clique, and each vertex of the root level is also a regular structure. Many practical applications follow these kinds of structures, including multicast trees, distributed hash tables, P2P applications, to name a few. The delay constraints for the query are randomly assigned from the 25-175ms range, which contains about 70% of the links in PlanetLab. Given that there usually thousands of mappings for these queries, the interesting measure here is the average time to find the first match. Figure 8 shows the results of this experiment.

The interesting observation 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 ECF algorithms. This reinforces the previously mentioned observation that in under-constrained queries and high-density graphs LNS is better suited to find the first solution.

### 6.4 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 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 time-out has expired. A partial set (subset) of all feasible embed-



**Figure 9. Probability distribution of the different types of results.**

dings is returned when the algorithm times out after finding some (but not necessarily all) feasible embeddings. Finally, the algorithm is said to have an inconclusive response, if it fails to produce any feasible embedding by the timeout. In the latter case, it is inconclusive whether or not a single feasible embedding exists.

Figure 9 shows the probability of each one of these results for all the experiments presented earlier. The probability of finding matches was over 70% for all cases. 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.

## 7 Related Work

The resource mapping problem has been extensively considered in the literature. As a representative of earlier works *matchmaking* [12] considered the problem in the simple scenario of finding a machine that matches the requirements of a job, a task that can be easily solved in linear time. The most recent evolution of this work, called *gangmatching*[13] considers the problem of coallocating a set of resources and jobs, subject to inter-dependencies that must be satisfied for a solution to be valid. This is closer to our *embedding problem*, but it does not include the network topology as a constraint. It could be argued that network

topology could be expressed as additional *classads* constraints, but by doing so, the size of the problem increases significantly making it much harder. Our algorithms were designed specifically to use the network as an additional constraint, helping reduce even more the solution space. It is worth noticing that our ECF algorithm pretty much resembles the basic idea of the *gangmatching* algorithm extended with our *candidate filtering* technique.

Another closely related work is Redline[6]. Redline redefines the *classads* language of *matchmaking* and poses the problem as a constraint satisfaction problem. Its algorithm is based on standard constraint satisfaction techniques using two phases: The first for node-constraint satisfaction and the second for arc-consistency propagation. However, this design assumes again that computing resources are individual objects that can be combined in any arbitrary way and, similarly to *gangmatching*, trying to incorporate network connectivity constraints would create an extra burden for this technique. Our LNS algorithm is constructed along the ideas of doing node and arc-consistency propagation, but using the underlying topology as an additional constraint to guide the propagation phase.

In [5] the authors define a new description language *vgDL* to describe sets of resources as compositions of aggregates with qualitative connectivity constraints. They also developed a search algorithm to find the mapping of the requested resources by the introducing several simplification rules to reduce the search space. Our work goes one step further by taking into account quantitative constraints and offering search algorithms that are free of false negatives. The importance of matching bandwidth requirements to meet a target QoS goal in MPI applications has been analyzed in [14].

A problem similar to ours that has been previously considered (for example in [4, 15, 9]) is the problem of optimizing the schedule of grid workflows in order to minimize the *makespan* of the application. Notice that our problem is complementary in the sense that the mapping solution provided by our techniques could be used as a candidate set for any of these schedule optimization techniques, therefore helping reduce the search space for the optimization problem.

Finally, it is worth noticing that many heuristics exist for constraint satisfaction problems, see for example [3, 10]. Our algorithms extend/adapt some of those techniques for the specific problem presented.

## 8 Conclusion

The work presented in this paper is one component of a framework for allocating resources in a distributed network subject to both qualitative and quantitative constraints. The mapping of the needed resources may not exist, may be unique, or there may be possibly many satisfactory mappings. NETEMBED is a service that lets applications identify those feasible embeddings as part of their resource selection process.

In its current state NETEMBED assumes OS-level isolation of the system's resources. As a future line of work NETEMBED will be aware of the load-dependence relationship between applications sharing the same resources and to take it into consideration to find feasible mappings. This case is of particular importance as in many real-life applications there are no mechanisms to guarantee this isolation. Take for example Planetlab slices running on the same machine, or internet links shared with many other applications.

## 9 Project Web Site

Additional information as well as a running demo of NETEMBED is available at: <http://csr.bu.edu/netembed>

## References

- [1] Cup. <http://www2.cs.tum.edu/projects/cup/>.
- [2] Jflex. <http://www.jflex.de/>.
- [3] S. Beale. *Hunter-Gatherer: Applying Constraint Satisfaction, Branch-and-Bound and Solution Synthesis to Computational Semantics*. PhD thesis, Scholl of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1997.
- [4] Y. Gong, M. E. Pierce, and G. C. Fox. Matchmaking scientific workflows in grid environments. In *Proceedings of IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2007)*, pages 19–21, Cambridge, USA, November 2007.
- [5] Y.-S. Kee, D. Logothetis, R. Huang, H. Casanova, and A. A. Chien. Efficient resource description and high quality selection for virtual grids. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 1*, pages 598–606, Washington, DC, USA, 2005. IEEE Computer Society.
- [6] C. Liu and I. Foster. A constraint language approach to matchmaking. *14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE'04)*, 00:7–14, 2004.
- [7] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. 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 Diego, CA, February 2004.
- [8] J. Londoño and A. Bestavros. NETEMBED: A network resource mapping service for distributed applications. Technical Report 2006-32, Boston University, December 2006.
- [9] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, B. Liu, L. Johnsson, and J. Mellor-Crummey. Scheduling strategies for mapping application workflows onto the grid. In *14th IEEE Symposium on High Performance Distributed Computing (HPDC 2005)*. IEEE Computer Society Press, 2005.
- [10] K. Marriot and P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, Cambridge, Massachusetts, 1998.
- [11] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, 2003.
- [12] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *IEEE International Symposium on High Performance Distributed Computing (HPDC98)*, pages 140–147, July 1998.
- [13] R. Raman, M. Livny, and M. Solomon. Policy driven heterogeneous resource co-allocation with gangmatching. In *International Symposium on High Performance Distributed Computing (HPDC03)*, pages 80–89, June 2003.
- [14] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. MPICH-GQ: Quality-of-service for message passing programs. In *Proc. SC00 (SC2000)*, Dallas, TX, November 2000.
- [15] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Proceedings of the 13th Heterogeneous Computing Workshop (HCW)*, Santa Fe, USA, 2004.
- [16] C. Yoshikawa. All-sites-pings for planetlab. <http://ping.ececs.uc.edu/ping/>.