Distributed Degree-Constrained Multicasting in Point-to-Point Networks

Fred Bauer

Anujan Varma

UCSC-CRL-95-09
March 3, 1995

Computer Engineering Department
University of California
Santa Cruz, CA 95064

E-mail: {fred,varma}@cse.ucsc.edu

Abstract

Establishing a multicast tree in a point-to-point network of switch nodes, such as a wide-area ATM network, is often modeled as the NP-complete Steiner problem in networks. In this paper, we present distributed algorithms for finding efficient multicast trees in the presence of constraints on the copying ability of the individual switch nodes in the network. We refer to this problem as the *degree-constrained multicast tree problem* and model it as the *degree-constrained Steiner problem* (DCSP) in networks. We consider two distinct approaches to the design of distributed DCPS heuristics. The first approach involves design of distributed versions of centralized DCSP algorithms. We introduce distributed versions of two DCSP heuristics: the *shortest path heuristic* (SPH) and the *Kruskal-based shortest path heuristic* (K-SPH). The second approach is to modify the solution obtained from an unconstrained heuristic to satisfy the degree constraints using a distributed post-processing algorithm. Unlike previous spanning-tree based approaches in which all nodes in the graph must participate, only nodes in the neighborhood of the multicast tree need participate in these algorithms. We compare the algorithms by simulation based on three criteria: competitiveness (the ratio between the sum of edge weights of the heuristic tree to that of the optimal tree), convergence time, and the number of unsolved networks. Our results show that each of the heuristics generated degree-constrained multicast trees within 10% of the best solution found. Surprisingly few test networks were unsolvable. The distributed post-processing heuristic presented is of particular interest since it may be used with any Steiner heuristic. When paired with a good unconstrained distributed Steiner heuristic, this post-processing heuristic gave away little competitiveness while converging rapidly.

Keywords: Degree-constrained multicasting, Steiner problem in networks, Distributed algorithms.

1 Introduction

Multicasting, defined here as the ability to connect multiple nodes in a point-to-point network by a low-cost tree, is likely to take an increasingly important role in data networks. Many existing networks already support multicasting. For example, the Internet MBONE service, a popular conferencing tool, already uses the multicast support recently added to the Internet [10]. Many emerging standards for packet-switched networks, notably ATM, Frame Relay and SMDS, include support for multicasting. Future applications such as audio and video conferencing, replicated database updating, and distributed resource discovery will rely on the ability of the network to perform multicast communication. Thus, multicasting will likely be an essential part of future networks.

The cost of multicasting quickly becomes unacceptable for many applications if a separate copy of data is transmitted from the source to each recipient. Transmitting common copies of data over a multicast tree is the preferred method. However, determining the optimal multicast tree for a network is a difficult problem. Previous authors have established that the multicast tree problem may be modeled as the *Steiner problem in networks* [3, 4, 5, 15, 33], referred to hereafter as the SPN, and that explicit solutions are prohibitively expensive. For example, two popular explicit algorithms, the *spanning tree enumeration algorithm* and the *dynamic programming algorithm* [33], have algorithmic complexities of $O(p^2 2^{(n-p)} + n^3)$ and $O(n3^p + n^2 2^p + n^3)$, respectively, where n is the number of nodes in the graph and p the number of multicast members. A number of good, inexpensive, centralized heuristics exist for the SPN and have been reviewed extensively elsewhere [5, 15, 20, 25, 26, 32, 33]. Some have been shown through analysis to produce solutions no worse than twice the optimal solution. [33]. That is to say, the sum of the edge weights of the heuristic tree is no more than twice the sum of edge weights of an optimal tree. In practice, our empirical evidence indicates that these heuristics find solutions much better than twice this bound with reasonable speed in most cases.

Most of the algorithms proposed in the literature for SPN are serial in nature. However, a few distributed heuristics have been known [8, 17, 22, 24]. Many of these algorithms are based on reducing the SPN to the minimum spanning tree problem (MST) and using a distributed minimum spanning tree algorithm such as the one described by Gallager, Humblet, and Spira [13]. The resulting minimum spanning tree is then pruned of unnecessary leaves and branches. For example, Chen, et al. [8] finds a Steiner tree by applying a distributed minimum spanning tree algorithm twice. First the algorithm is applied to the original graph. This first minimum spanning tree is used to create a *shortest path forest* composed of disjoint trees and edges that together form a subgraph of the original graph. The distributed minimum spanning tree algorithm is then applied a second time to this subgraph. The solution is obtained by pruning unnecessary leaves and branches from this second minimum spanning tree. Likewise, Kompella, et al. [22] describe two distributed versions of earlier centralized heuristics proposed by the same authors [21]. Both of these distributed heuristics first build a *constrained Steiner tree* that reflects the combined criteria of cost and delay. Gallager, Humblet and Spira's distributed MST algorithm is then applied to this constrained Steiner tree and the solution tree is pruned. The two heuristics differ in their criteria for choosing edges when building the MST. The primary disadvantage of using a minimum spanning tree algorithm is that every node in the network must participate in the search for a Steiner tree. This may not be feasible or desirable in practice. In contrast, Jiang's Steiner heuristic [17] does not use an MST algorithm, but instead relies on flooding from the multicast source to find the shortest path tree to the multicast members. The source node repeatedly broadcasts messages and builds a Steiner tree from responses received until a suitable tree

is found.

Finding a multicast tree is complicated by the likely heterogeneous nature of the multicast environment. The routers or switches in an internetwork will likely vary in their ability to support multicasting. Some nodes may not support multicasting; others may be limited in the number of multicast copies they can reasonably make [6, 7, 19, 34]. The multicast capability of each node is represented in this paper by a degree-constraint. Thus, a degree constraint of d implies that the corresponding switch or router is able to forward copies of an incoming packet to a maximum of $d-1$ output ports. The problem of finding a multicast tree in the presence of copy constraints in individual switches is referred to as the *degree-constrained multicast tree problem*. We model the degree-constrained multicast tree problem as the *degree-constrained Steiner problem in networks* (DCSP), first described by S. Voss [31]. Formally, the DCSP as used in this paper is defined as follows.

GIVEN: A simple, undirected, connected graph $G = (V, E)$ with n nodes, non-negative edge cost $c_{i,j}$, p multicast members $Z \subseteq V$, and node degree constraints $k_i \geq 2$.

FIND: A multicast tree T such that the degree of each node $d_i \leq k_i$ and the sum of its edge weights is minimum among all possible choices of the tree satisfying the degree constraints.

The degree-constrained Steiner problem in networks is of particular interest to ATM networks since multicast copies of ATM cells must be made quickly, typically by hardware. Whereas a low-speed switch or router might reasonably approximate infinite copy-capability, high-speed switches may have limited copy capabilities due to their speed constraints. Even when the switches allow multicasting to an arbitrary number of destination ports, there are several advantages in limiting the number of copies made by each switch. For example, some packet-switch architectures implement multicasting by circulating copies of packets through the switch fabric multiple times [28]. Thus, keeping the degree small reduces the number of passes needed through the switch fabric. In addition, a degree-constrained multicast tree also distributes the load more evenly among the nodes in the network than an unconstrained tree. This has two benefits: (i) the task of making multicast message copies is shared among more nodes, and (ii) the damage inflicted on the tree by the failure of a single node is reduced.

The degree-constrained Steiner problem in networks is a relatively new problem [16, 27, 29, 30]. The degree-constrained Steiner problem is NP-complete [30] and contains the NP-complete problem of determining a degree-constrained spanning tree [12, 18]. Furthermore, finding a solution to the DCSP is also NP-complete [30]. In practice, however, the heuristics rarely failed to find a solution in our test networks.

Few centralized DCSP heuristics exist in the literature [27, 30] and we know of no published distributed DCSP heuristics. Since the degree-constrained spanning tree problem is also NP-complete, distributed Steiner heuristics that use a distributed minimum spanning-tree algorithm cannot easily be modified for DCSP.

In an earlier paper, we introduced several centralized DCSP heuristics and compared their runtime and the competitiveness of the solutions produced [1]. The competitiveness of a solution improves when a tree has lower cost where cost is the defined to be sum of the tree edge weights. This paper introduces and analyzes distributed algorithms for DCSP. We consider two distinct approaches to the design of distributed DCPS heuristics. The first approach involves design of distributed versions of centralized DCSP algorithms. We introduce distributed versions of two DCSP heuristics K-SPH and SPH described in [1]. The second approach is to find a Steiner tree first with no degree constraints and then modify the solution to satisfy the degree constraints using a distributed post-processing algorithm. The second approach

has the advantage that any of the known distributed Steiner heuristics can be used during the first step; for the second step, we present a distributed "fixup" heuristic to construct a degree-constrained multicast tree from the unconstrained one. The two approaches are compared by simulating the algorithms on sparse, point-to-point, networks of switches with varying multicast capacity. We analyze the message and time complexity of the algorithms, and compare them on the basis of three criteria: competitiveness, convergence time, and the number of unsolved networks. Ideally, a heuristic's competitiveness is measured by the ratio of its tree cost (sum of edge weights) to that of an optimal tree. This proves to be impractical, since computing the latter may be prohibitively expensive in large networks. Instead of an optimal tree we use the best heuristic solution found for each network by any heuristic we have tested, distributed or centralized.

Our simulations of the algorithms are performed on a large set of sparse, randomly-generated network topologies. We restrict our analysis to sparse networks for two reasons: (i) they are more representative of real point-to-point networks, and (ii) they are inherently more difficult to solve because fewer solutions exist in a sparse network than a dense one. Similarly, the simulated multicast groups are small relative to the size of the network, reflecting likely multicast applications. Note that our results are not specific to any particular type of network such as ATM, but may be applied to any point-to-point network matching these assumptions.

The results of our research show that the distributed versions of heuristics K-SPH and SPH compare favorably with their centralized versions in the competitiveness of the solutions produced. Despite each node having only local topology information, the distributed versions of these programs often produced trees within 10% of the best solution found by any heuristic and solved over 90% of our test networks. However, the most interesting results concern the post-processing heuristic. This heuristic showed rapid convergence and produced degree-constrained trees from unconstrained ones while sacrificing little of their quality.

The remainder of this paper is organized as follows. Section 2 summarizes previous DCSP heuristics of interest, introduces distributed versions for two of these heuristics, and analyzes their convergence-time and message bounds. Section 3 presents a new distributed post-processing algorithm that can be used to modify any valid Steiner tree solution to satisfy the given degree constraints. Section 4 compares the algorithms by convergence time, competitiveness, and the number of unsolved test cases. Finally, Section 5 concludes the paper with a discussion of the results.

2 Degree-Constrained Steiner Heuristics

In this section, we summarize previous degree-constrained Steiner heuristics and introduce two distributed degree-constrained Steiner heuristics.

Before continuing, we make the following basic definitions and notations. Z is the set of multicast destinations, S is the set of non-multicast nodes $V - Z$, $P_{i,j}$ is the shortest path between nodes i and j, and $d_{i,j}$ is the distance of the shortest path between nodes i and j. Graph distances will be defined as follows: The distance between two nodes is the distance of the shortest path between them. Likewise, the distance between a node and a tree is the distance of the shortest path between the node and any node in the tree. Finally, the distance between two trees is the distance of the shortest among all paths between any node in one tree and any node in the other tree. As in [13], we append the weight of an edge or path with the index of its destination node in determining shortest paths so that, in case of a tie, the actions of the individual nodes would be consistent. Since we do not allow multiple edges between node pairs, this ensures that all the nodes select the same edge or path, given the same set of edge weights.

2.1 Centralized Degree-Constrained Steiner Heuristics

To be suitable for distributed implementation, a DCSP heuristic must satisfy four criteria. It must (i) use the existing routing information available at each node in the network, (ii) use minimal computational and network resources, (iii) require a minimum of coordination between neighbors, and (iv) limit itself to nodes directly involved in the multicast. Of the centralized DCSP heuristics evaluated in [1], we chose the following four heuristics as candidates for distributed implementation: the *shortest-path heuristic* (SPH), a variant of SPH known as SPH-Z, the *Kruskal-based shortest-path heuristic* (K-SPH), and the *Average distance heuristic* (ADH). Each heuristic's unconstrained version is described in [25] and each heuristic's degree-constrained version is described in [1]. A brief summary of each degree-constrained heuristic follows.

DCSP SPH

Heuristic SPH, whose unconstrained version was introduced in [26], initializes the multicast tree to an arbitrary multicast member. It then grows the tree by successively adding the next closest multicast member to the multicast tree by the shortest, degree-constrained path between the multicast member and the tree. The algorithm terminates when all the members have joined the tree or the remaining multicast members cannot be connected.

DCSP SPH-Z

Heuristic SPH-Z, whose unconstrained version is introduced in [25], is a variant of SPH that runs SPH repeatedly, once for each multicast member (Z-node). This results in up to Z degree-constrained multicast trees. At SPH-Z's conclusion, the lowest cost tree is returned as the result.

DCSP K-SPH

Heuristic K-SPH, whose unconstrained version was introduced in [23], differs from SPH in the manner in which tree is expanded. Instead of growing the tree one node at a time, the algorithm joins subtrees pairwise repeatedly until all the multicast nodes are part of a single tree or components can no longer be connected. The algorithm starts with Z subtrees, each a multicast member itself. In the expansion step, it finds two subtrees that are closest to each other and joins them along the shortest, degree-constrained path between them to form a single subtree.

DCSP ADH

Heuristic ADH, whose unconstrained version is described in [25], is a generalization of K-SPH. The degree-constrained version repeatedly connects the two closest sub-trees using the *most central* node. The most central node is defined to be the node with the least average distance to all multicast members. ADH terminates when a single tree remains spanning all multicast members or the remaining components cannot be connected.

2.2 Distributed Degree-Constrained Steiner Heuristics

After further consideration, only two of the four heuristics, SPH and K-SPH, remain as suitable candidates for distributed implementation. Both heuristics SPH-Z and ADH fail our criteria for conversion. Although

heuristic SPH-Z appears initially attractive as a distributed heuristic, its component distance and degree-constraint information for each of the Z instances could be distinct. This forces as many as Z copies of component information at each node and a virtual storm of network messages before convergence. Likewise, ADH fails our criteria because its calculation of the most central node requires excessive overhead for coordination between nodes in the network. In addition, our earlier results indicate that, on the average, the solutions produced by K-SPH and ADH are of nearly identical quality. Of these two heuristics, K-SPH is the more attractive candidate because of its relative simplicity and lower running time.

Distributed heuristics SPH and K-SPH are designed to run as asynchronous, independent processes running one per node in a degree-constrained network. Each distributed heuristic assumes that the routing tables in each node is up-to-date; no topology changes occur during the execution of the algorithm; the network is connected; every node has a unique index; each multicast member has knowledge of the indices of all other multicast members; each multicast is able to determine the distance to each of them from its routing table; and all non-leaf nodes have a degree-constraint ≥ 2.

Heuristic SPH is inherently a serial algorithm, since there is only one subtree expanding itself at any time during the execution of the algorithm and nodes must join the tree serially. Heuristic K-SPH, on the other hand, allows many of the join operations to proceed in parallel. The latter, however, is substantially more difficult to parallelize because of the significant amount of coordination that may be needed while combining subtrees. In the following, we present distributed K-SPH first, followed by a similar distributed implementation of SPH.

2.2.1 Distributed Heuristic K-SPH

Like its centralized version, distributed K-SPH starts with a forest of Z multicast members (Z-nodes) and connects them pairwise into successively larger subtrees until a single multicast tree remains or no further connections are possible. We refer to the subtrees during the execution of the algorithm as *fragments*. Thus, at the beginning of the algorithm, there are Z fragments, each a trivial subtree consisting of one Z-node.

At any instant during the execution of the algorithm, each node in the network is either part of a fragment or has not been yet been included in the multicast tree. Note that every Z-node is always a fragment node and every non-member node (S-node) is initially a non-fragment node. When two fragments merge, the nodes in both fragments and the nodes in the path connecting them become the fragment nodes of the new, merged fragment.

Each fragment has a *fragment leader* coordinating the activities of the fragment. This fragment leader is the fragment Z-node with the lowest index. Each fragment leader executes the same finite state machine shown in Figure 1. Other fragment node executes a simplified version of the leader's finite state machine shown in Figure 2. Initially, each multicast member is the leader of its own one-node fragment; when two fragments merge, leadership is assigned to the fragment leader with the lower index. To identify fragments uniquely, each fragment has the same index as its leader and each fragment node is aware of its fragment index.

At the highest level, each fragment, guided by its leader, executes the pseudocode shown in Figure 3. During the execution of the algorithm, each fragment attempts to merge with its closest neighboring fragment. This is accomplished in two steps — a *discovery* step and a *connection* step. During the discovery step, the leader gathers and updates its information on other fragments and graph nodes. Based on the information gathered, it determines the closest fragment to merge with. During the connection step, it communicates with the closest neighbor fragment's leader, requesting a merge. This closest fragment leader is simply the

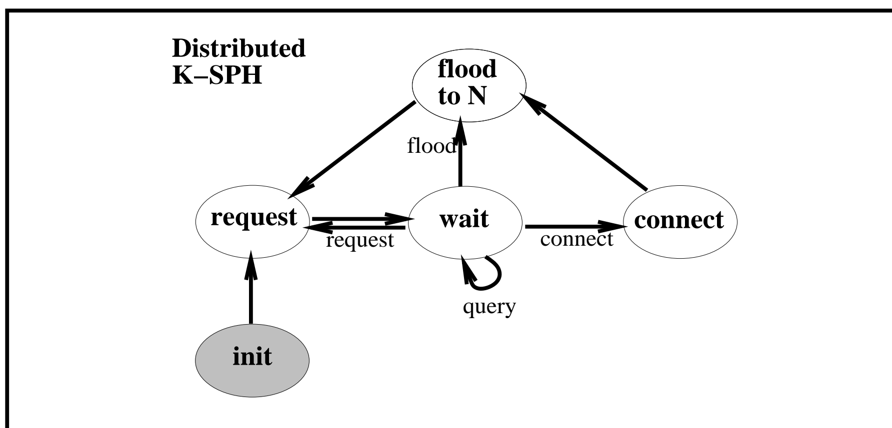

Figure 1: The finite state machine for fragment leaders.

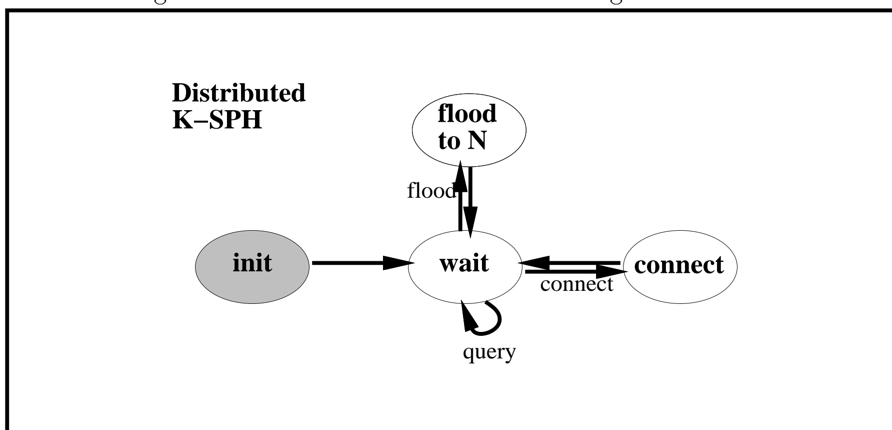

Figure 2: The finite state machine for other fragment nodes.

Z-node with the same index as the closest fragment. If accepted, the leader with the lowest index attempts to connect the two fragments. Regardless of the outcome (the request is rejected, the subtrees are connected, or the connection attempt fails), the cycle repeats until the algorithm terminates.

Distributed K-SPH processes running on each node rely on shortest path and degree-constraint information available at its node, as well as information maintained by the fragment leaders. The shortest path information stored at each node is the distance, next hop and next-to-last hop of the shortest path to other nodes. This path information is similar to that stored by distance-vector routing protocols and may already be available in each node's routing tables. If so, distributed K-SPH may use the existing tables, avoiding unnecessary extra storage at each node. If not, this information may be derived using a distance vector routing algorithm such as [14]. The next-to-last hop table allows distributed K-SPH processes to derive the entire shortest path between nodes by recursively considering the path as described in [9]. Each node also stores its degree-constraint information and the index of its fragment. Initially, only multicast nodes have a fragment index (its own index). Each leader maintains additional shortest path and degree-constraint information for its fragment. This information augments the shortest path information at each node. For example, the leader stores only the distance, and the head and tail of the shortest path between its fragment

```
while fragments remain and valid paths exist do

    # algorithm discovery step
    update fragment information for nearby fragments
    f ← closest fragment

    # algorithm connection step
    request merger with fragment f
    if request accepted then
        attempt connection to fragment f
        if connection established then
            merge fragments
        else
            restore original fragments
        end if
    end if
end while
```

Figure 3: Pseudocode for fragment leaders.

and every other fragment. The additional details necessary to build the path between fragments is stored at the head of the path, a node in the leader's subtree (Note that the shortest path between fragments need not start or end at a leader node). The leader also stores the *currently available* degree-constraint information for every node in the graph. Initially, each leader knows only about its own degree-constraint and assumes all other degree-constraints to be infinite.

When distributed K-SPH starts, each Z-node, the leader of its own trivial one-node fragment, already knows its distance to every other fragment as provided by the initial distance tables and no discovery step is necessary. Instead, each distributed K-SPH leader starts with the connection step, described as follows.

The connection step During the connection step, each leader attempts to connect its fragment with the closest fragment, known as its *preferred fragment*. It does so by sending a *merge request* message to the leader of the preferred fragment (That is, the Z-node with the same index as the preferred fragment). A leader receives one of three responses to its request: *accept*, *reject*, or *busy*. We consider each response in turn below.

The busy response occurs when a fragment's request arrives at its preferred fragment while the latter is in its discovery step described below. While in the discovery step, a leader cannot accept or reject merge requests, as it is in the process of updating its information. Instead, the busy response is sent. When the requesting leader receives the busy response, it repeats its request in the hopes or reaching its preferred fragment after its discovery step. A leader will repeat its connection request until it receives either an accept or reject response.

When a leader receives a connection request from a fragment other than its preferred fragment, it returns a reject message. This message forces the requesting fragment into a discovery step to find another preferred fragment. If a former leader node receives a connection request from any fragment, it returns a reject message since a connection is no longer possible to the old fragment.

```
# request merger with fragment f
do
      send request to fragment f's leader
      wait for response
      update degree-constraint information from responding fragment
until accept or reject

leader ← this fragment index < fragment f's index
if accept and leader then
      # attempt connection to fragment f
      send connect message to head of shortest path

      wait for connection success or failure
      if failure then
            send reject to fragment f's leader
end if
```

Figure 4: Fragment leader pseudocode for the connection step.

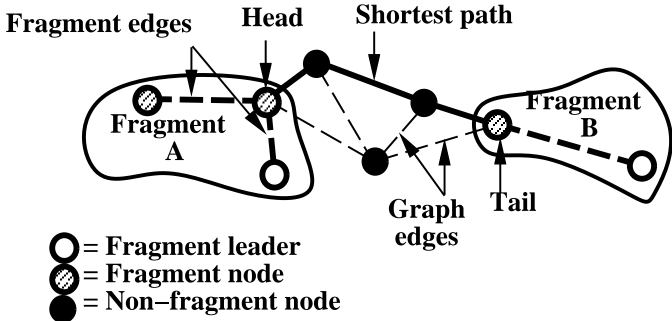

Figure 5: Example of fragments A and B merging.

Figure 4 shows the pseudocode for the connection step.

When two fragments exchange merge requests with one another, each responds by returning an *accept* message. Once an accept message is sent, the fragment may not leave the connect step or accept a request from another fragment until the connection attempt completes. Of the two leaders in a connection attempt, only the leader with the lower index acts, while the leader with the higher index waits passively for the result of the connection attempt. This is because if the connection attempt succeeds, the leader with the lower index becomes the leader of the new, merged fragment. The leader with the lower index initiates a connection attempt by sending a message to the head of the shortest path between the two fragments, a node in its fragment. In its message to the head of the shortest path, the leader specifies the tail of the shortest path, a node in the other fragment. Upon receiving the connect message, the head node sends a connect message along its shortest path to the tail node.

When two fragments A and B merge as shown in Figure 5, the shortest path used to join them must

have its head in one fragment, its tail in the other, and pass through only non-fragment nodes.

The connect message may either reach the target fragment or be *blocked*; blocking may occur either as a result of the degree constraint being exceeded in one of the intermediate nodes, or because the message reaches a node in a third fragment before reaching the target fragment. In either case, a status message returns to the head of the shortest path.

Consider the case of a successful connection first. In this case, the connect message travels down the shortest path, reserving intermediate nodes in the path as part of the new fragment, until it reaches a node in the target fragment. It is possible that the first node reached in the target fragment is not the specified tail. This occurs when the leader's shortest-path information for other fragments may be stale and an intermediate node in the selected path is already part of the other fragment. In any case, the connect message stops at the first node in the target fragment it reaches. The target fragment node then sends a status message back along the shortest path to the head of shortest path. Each reserved, non-fragment node along the path receives the status message, includes itself in the new, merged fragment, and passes the status message along the path. The head of the shortest path forwards the status message to its leader, now the leader of the new, merged fragment. This completes the connection step and the leader enters the discover step described below.

Now consider the case of an unsuccessful connection. In this case, the shortest path between the fragments is blocked. This could occur because a node in the shortest path has already reached its degree constraint, or has become part of a third fragment. When the connect message reaches a blocked node, the blocked node returns a status message along the shortest path to the head of the path. As each intermediate node receives the status message, it removes its reservation from the new fragment, becoming an non-fragment node once again. The head of the shortest path forwards the status message to the leader. The leader informs the other fragment leader of the connection failure by sending a reject message. This completes the connection step. Both leaders then enter the discover step described below.

States *request, wait* and *connect* in Figure 1 comprise the connection step.

The discovery step The discovery step accomplishes three tasks: (i) it informs every node in the fragment of its new fragment index, (ii) it gathers fragment and degree-constraint information about nodes close to the fragment, and (iii) it refreshes its information on shortest paths to other fragments. The pseudocode for the discovery step is shown in Figure 6. Each fragment leader achieves these tasks by performing a multicast on its fragment rooted at itself. In the multicast message, the leader includes the fragment index, the distance to the preferred fragment, degree-constraint information, and shortest paths to other fragments. As each node in the fragment receives the multicast, it updates its fragment index, queries nearby nodes and passes the multicast message to its children. Only those nodes that lie within the distance from this fragment to the preferred fragment are queried for fragment index and degree-constraint information. The objective of queries to nearby nodes is to find fragment nodes closer than those already known by the leader. Figure 7 illustrates a case where this is useful. Fragment B's leader believes that fragment C is the closest fragment. During fragment B's discovery step, it instructs fragment nodes to query those nodes closer than fragment C. This distance is the distance between node 3, the head of the path to fragment C, and node 4, its tail, and is marked by the dotted circles around each of fragment B's nodes. Since nodes 1 and 2 fall within one such circle, they receive queries and fragment B's leader discovers the closer fragment A. Queries could be sent to all nodes in the graph, but are limited to nodes within a small distance for two reasons: (i) a set distance avoids broadcast storms and (ii) new shortest paths discovered should be shorter than those already

```
# send all children update request
for all fragment children do
    send fragment index, closest fragment distance, shortest path information,
        and degree-constraint information to each child
end for

# query all nodes closer than closest fragment
for nodes nearer than closest fragment do
    send nodes query for component and degree-constraint information
end for

wait for responses
update shortest path information, and degree-constraint information

# forward results
if not leader then
    send summary of shortest path information and degree-constraint
        information to parent
end if

if leader then
    f ← closest fragment
```

Figure 6: Fragment leader pseudocode for the discovery step.

available. The discovery step is implement by state *flood-to-N* in Figure 1.

Analysis of Distributed K-SPH Algorithm: Having described the distributed K-SPH in the previous section, we now turn to its properties. We use a directed *request graph* to show the relationship of fragments to one another during the execution of the algorithm. Each fragment in the network is represented by a node in the request graph and its current choice of preferred fragment by a directed edge. Figure 8 illustrates an example graph with three vertices representing fragments A, B, and C. In this example, the fragment pair A and B request each other, while a third, more distant fragment C requests fragment B. Fragments A and B will merge, creating a new fragment that will form a pair with fragment C and merge. A fragment is considered *stable* when it is in the states wait or request since its choice of preferred partner is unknown when the fragment is in states connect, flood-to-N, or init.

The request graph can be used to show that the distributed K-SPH algorithm does not deadlock. To prove that the algorithm will terminate, we need to only show that at any time during the execution of the algorithm, the shortest-path distances maintained by two of the fragments to each other will converge to the same value in a finite time (that is, a cycle of length 2 in the request graph). These two fragments will then merge to form a new fragment. Thus, by induction, the algorithm will terminate in finite time.

Property 1 *At any time during the execution of the algorithm, the shortest-path distances maintained by two stable fragments to each other will converge to the same value within a finite time.*

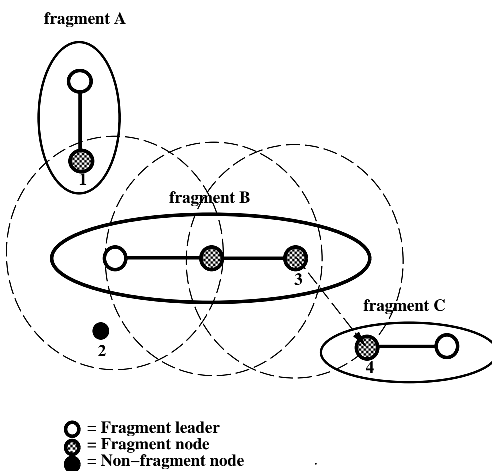

Figure 7: Illustration of the querying process in the discovery phase of distributed K-SPH.

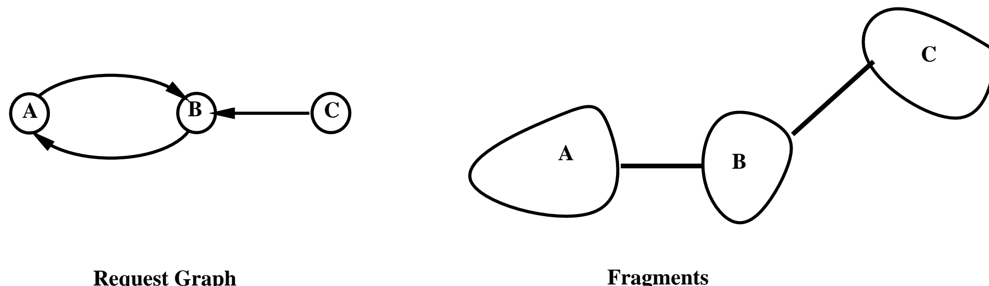

Figure 8: An example for request graph.

Proof: Initially, the shortest path between any two single-node multicast members is the shortest path between their fragments. This path is, by definition, equal. Suppose that at some later time, the shortest paths between two fragments differ in distance as shown in Figure 9. Suppose further, that one fragment, in this case fragment B, has the longer path. The next time fragment B enters state flood-to-N, it will query every node in its neighborhood for fragment and degree-constraint information. Any node in fragment A closer to B must fall within B's neighborhood and will become the tail of a new, shorter path to fragment A. The shortest of all such paths will become fragment B's new shortest path to fragment A. By a symmetrical argument, the paths between fragments A and B must converge on the same distance.

Inconsistencies may also occur when a path is blocked. Assume that only one of a pair of fragments finds that the shortest path between them blocked. Assume fragment A has a shortest path to B, but B's shortest path to A is blocked. The shortest path between fragments can only be blocked by a degree-constrained node, or by a node belonging to a third fragment C. If the shortest path is blocked by a degree-constrained node, fragment A will know of the over-constrained node the next time it exchanges degree-constraint information with fragment B (that is, when it receives a reject response to its merge request). If the shortest path is blocked by a third fragment C, fragment A will query the blocking node in fragment C the next time fragment A enters the discovery phase (state flood-to-N). In addition, C will also update its distance to A during its recovery phase, resulting in consistent values for the distance between A

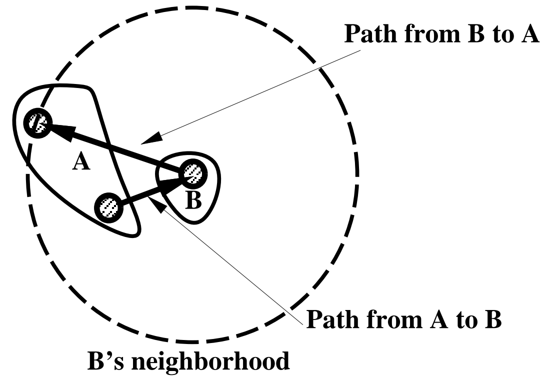

Figure 9: Two subtrees with different shortest paths.

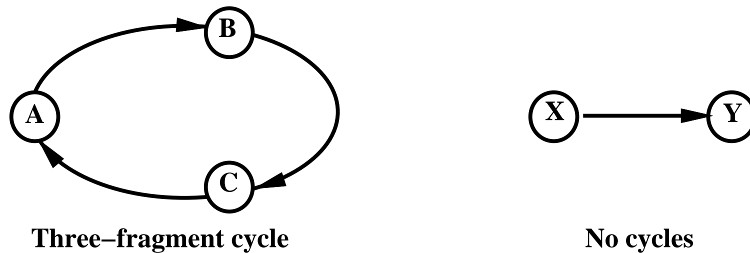

Figure 10: A request graph demonstrating deadlock.

and C.

A deadlock occurs when no two-fragment cycle exists in the request graph even when all fragments are stable. Figure 10 shows two such examples. In the first case, three fragments are locked in a cycle and in the other, one fragment has no outgoing edges and cannot merge with any other fragment. Either of these cases could mean that distributed K-SPH would never terminate. In the following, we show that such deadlocks cannot occur.

Property 2 *Distributed K-SPH does not deadlock.*

Proof: Let $d(I, J)$ represent the distance between fragments I and J. In Figure 10, stable fragments A, B and C are locked in a three-node cycle. Since each fragment prefers the closest fragment, the following inequalities must hold: $d(A, B) < d(A, C)$, $d(B, C) < d(B, A)$, and $d(C, A) < d(C, B)$. However, we know from property 1 that at least two of the fragments, say A and B, must have equidistant shortest paths. This leads to a contradiction. A similar argument holds for any cycle of more than two fragments. Consider the case where no cycle exists as shown by fragments X and Y in Figure 10. Fragment Y has no outgoing edge, which indicates that it has no shortest path to any fragment. This means that Y's path to X must be blocked and by property 1 will eventually be discovered. Distributed K-SPH will terminate when both X and Y have no outgoing edges.

Convergence Time and Number of Messages: We now derive some simple asymptotic bounds on the number of messages and convergence time of the distributed K-SPH algorithm. We consider the both the degree-constrained and the unconstrained versions of the algorithm.

Distributed K-SPH uses the least number of messages for both the unconstrained and constrained cases when the network has $Z = 2^i$ multicast nodes, any number of non-multicast nodes, and fragments always find a partner. Under these conditions, a total of $\frac{Z}{2^1} + \frac{Z}{2^2} + \cdots + \frac{Z}{2^i} = Z\frac{2^i-1}{2^i} = 2^i\frac{2^i-1}{2^i} = 2^i - 1$ merges occur during i rounds. Each fragment merges using a relatively small number of messages and the new fragment enters the discovery phase, In the discovery phase, each fragment node queries every child and neighbor for fragment and distance information. Assume that on average each fragment node queries a finite number of neighbors and children approximated by c. The total number of messages sent by multicast members during each round is $cZi = cZ \log Z = \Omega(Z \log Z)$. During each of the $\log Z$ rounds, the longest round-trip message time between leader and fragment root dominates. This round-trip time can be at worst twice the diameter of the graph, and at best a constant. Thus, the time to converge is lower-bounded by $c \log Z = \Omega(\log Z)$.

Now consider the worst case with no degree constraints. In the worst case, only one fragment finds a partner during any round. Thus, $Z - 1$ rounds occur before a solution is found. If fragments are always relatively large then the number of messages would be the number of rounds times the number of nodes on all fragments, $c(Z - 1)N = O(ZN)$. If the round-trip times during each of the $Z - 1$ rounds is large and close to twice the graph diameter, $2D$, then the convergence time for this case is $2D(Z - 1) = O(DZ)$.

The worst case for degree-constrained distributed K-SPH is similar except that each pair of fragments might try many paths between them before finding a viable, degree-constrained shortest path between them. The number of alternate shortest paths between nodes could be as high as the number of fragment nodes in the initiating fragment. Thus, if the initiating fragment is always large, each of the $Z - 1$ rounds could iterate as many as N times. The number of messages for this case would be N time greater than the unconstrained worst case, that is, $O(ZN^2)$. Likewise, the convergence time would be the number of rounds (now $\leq (Z - 1)N$) times the largest possible round-trip time, or $O(DNZ)$.

These bounds are summarized in Tables 1 and 2, along with the bounds for the other algorithms considered in this paper. Our results from simulations of the unconstrained version of the algorithm show that the rates of increase of both the convergence time and the number of messages with the number of nodes fell within these bounds [2].

2.2.2 Distributed SPH

The distributed shortest path heuristic is a special case of distributed K-SPH described in section 2.2.1. In distributed SPH, any one of the multicast members may act as the source of the multicast, referred to here simply as the *source node*. In contrast to distributed K-SPH, only one fragment, the *source fragment* grows, connecting multicast members to itself until all the multicast members are part of the same fragment. The heuristic terminates when a single tree remains or the source fragment cannot merge with the remaining fragments due to degree constraints.

In SPH, the preferred fragment of every fragment is always the source fragment. The sole exception, of course, is the source fragment itself which prefers its closest fragment. Using the same connection step outlined for heuristic K-SPH, the source fragment merges with its closest fragment. As the source fragment grows, it uses the same discovery step to determine the new, closest fragment. The source fragment never changes its index. This preserves the source fragment's original index so that non-source fragments never need to change their preferred fragment index. As a consequence, non-source fragments do not enter the discovery phase. In all other respects, distributed SPH is very similar to distributed K-SPH.

Figure 11 shows the finite state machine used by each node.

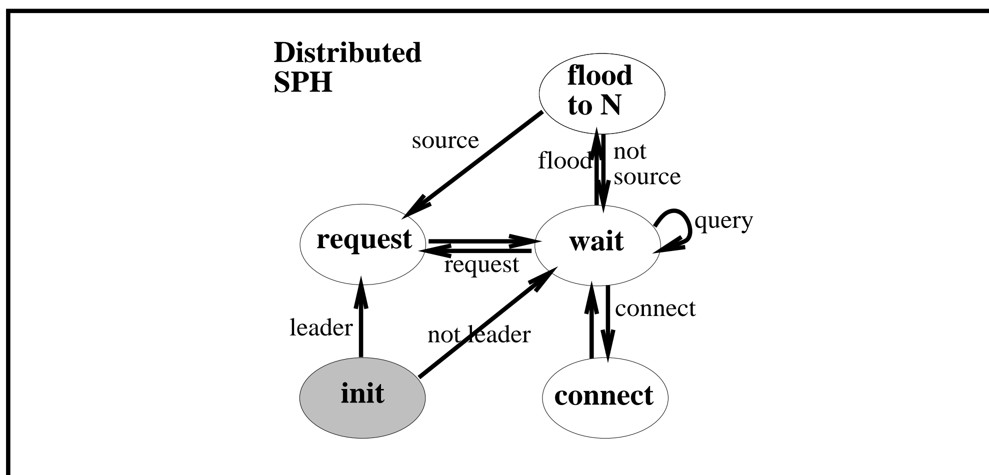

Figure 11: The finite state machine for each node performing the distributed SPH heuristic.

Algorithm Analysis: Since distributed SPH is a special case of distributed K-SPH, its analysis proceeds similarly to that of distributed K-SPH. For example, it too will not deadlock as shown by property 3.

Property 3 *Heuristic SPH does not deadlock.*

Proof: Consider Figure 10 again. A three-node cycle such as the one in Figure 10 cannot occur in distributed SPH because every fragment except the source prefers the source fragment. Thus, the longest request graph cycle in distributed SPH has length two: an edge from the source to a fragment and the return edge. A longer request cycle is an error. A zero-node cycle, however, indicates either an error or heuristic termination. Since every fragment except the source fragment always prefers the source fragment, only a source fragment can break the two-node cycle it has with every fragment. However, this is also a condition for termination. In the degree-constrained case, this is a valid termination condition.

Messages and Convergence Time: Like distributed K-SPH, distributed SPH uses the least number of messages when $Z = 2^i$ multicast members exist. Distributed SPH differs from distributed K-SPH in that only two fragments merge during a round. Assume that on average each fragment node queries a finite number of neighbors and children approximated by c. The number of messages in this case would be $c + 2c + 3c + \cdots + (Z-1)c = c\frac{(Z-1)^2-(Z-1)^2}{2} = O(Z^2)$. The round-trip time during each of the $Z-1$ rounds cannot be greater than twice the graph diameter and the convergence time is $c(2D)(Z-1) = \Omega(DZ)$.

In the unconstrained worst case, assume that the source fragment grows quickly and the round-time distance for messages approaches twice the graph diameter, $2D$. The number of messages in this case is $c(Z-1)N = O(ZN)$. Likewise the convergence time becomes $2D(Z-1) = O(DZ)$.

In the degree-constrained worst case, just as in distributed K-SPH the source fragment might try many paths before finding a viable, degree-constrained shortest path. The number of alternate shortest paths between nodes could be as high as the number of fragment nodes in the source fragment. Since the source fragment is always the largest fragment, the number of alternate paths available grows quickly. The number of messages for this case would be N time greater than the unconstrained worst case, or $O(ZN^2)$. Likewise, the convergence time would be the number of rounds (now $\leq (Z-1)N$) times the largest possible round-trip time, or $2DNZ = O(DNZ)$.

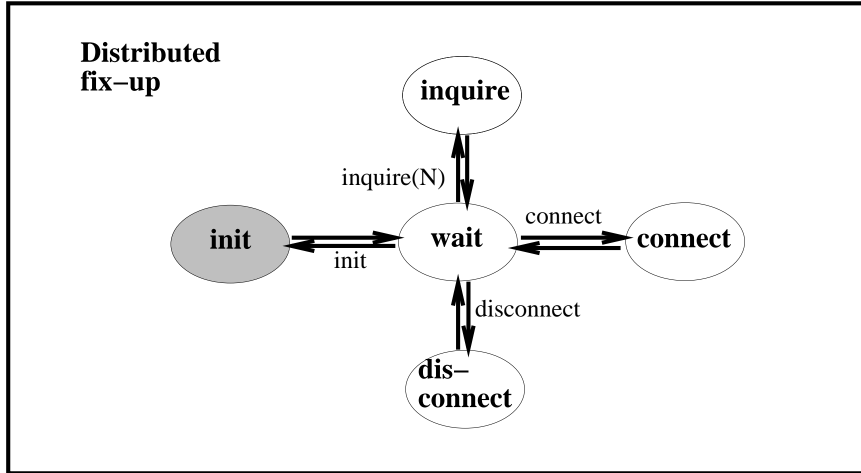

Figure 12: The finite state machine for each node performing the distributed Fixup heuristic.

These bounds are summarized in Tables 1 and 2, and have been found to agree with results from simulation of the unconstrained SPH heuristic [2].

3 Post-Processing Heuristics

An alternate approach to using a distributed algorithm for finding a degree-constrained Steiner tree is to first construct a Steiner tree with no degree constraints and then modify the solution to satisfy the constraints. This has the advantage that any known distributed Steiner heuristic can be used in the first step. If the number of nodes violating their degree constraints in the solution is small, the second step can typically be completed in much less time as compared to the first. The second step involves removing incident edges from the Steiner tree and finding alternate paths to re-connect the fragments such that the degree constraints are not violated. In this section, we describe a distributed heuristic to find a degree-constrained Steiner tree using this approach.

3.1 Distributed Heuristic Fixup

Heuristic Fixup modifies an unconstrained tree by examining each vertex in the unconstrained tree for degree-constraint violations and replacing the edges of an over-constrained node, one at a time, with an alternate degree-constrained path. In Figure 13, the edge between the over-constrained node and fragment A is replaced by an alternate path. This alternate path is discovered by exploring alternate paths between neighbors of the over-constrained node. It is reasonable to expect competitiveness and the number of solved cases to improve with the size of the neighborhood searched. While our results confirm this intuition, they also demonstrate that a point of diminishing returns exist. We use three hops for our neighborhood size because it offered the best balance between convergence time and competitiveness. The effect of neighborhood size on heuristic Fixup is discussed in Section 4.

Distributed heuristic Fixup first runs on the tree's leader and examines the tree, looking for degree-constrained nodes. The algorithm assumes that the leader knows members of the multicast tree and their degree constraints. Note that this information may be gathered by the leader by propagating a request

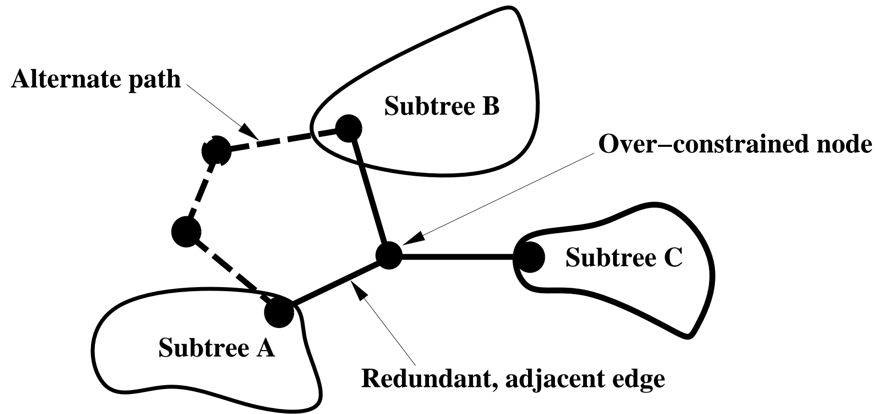

Figure 13: Replacing an adjacent edge to an over-constrained node.

along the multicast tree. In our test cases, a small but significant number of trees generated by distributed heuristic K-SPH were already degree-constrained. Of course, a tree that is already degree-constrained need no further processing. If, however, over-constrained nodes exist, heuristic Fixup runs on each over-constrained node in sequence, replacing one edge at at time until the degree-constraints are met. Heuristics Fixup terminates when either a degree-constrained tree is found or no alternate path to an over-constrained node is available. Heuristic Fixup runs on one over-constrained node at a time to reduce algorithm and communication complexity. The finite state machine used by heuristic Fixup's nodes is shown in Figure 12.

Just as in distributed heuristics SPH and K-SPH, heuristic Fixup relies on shortest path and degree-constraint information available at each node. The shortest path information stored at each node is the distance, next hop and next-to-last hop of the shortest path to graph nodes. In addition, heuristic Fixup passes each node the degree-constraints of every other node in the tree. The shortest path information allows each node to determine intermediate nodes in a shortest path while degree-constraint information allows nodes to eliminate over-constrained paths. Since heuristic Fixup runs on only one node at a time, changes in degree-constraint information made during heuristic Fixup are easily recorded and are passed on to neighborhood nodes.

Consider the fragments left if an over-constrained node were deleted from the tree. In Figure 13, three such fragments, fragments A, B, and C would remain. At each over-constrained node, heuristic Fixup asks nodes in its neighborhood for alternate degree-constrained paths to connect two such fragments, replacing one of the over-constrained node's edges. Such an alternate path must start at one fragment's node, traverse non-fragment nodes and end at another fragment's node. Using its shortest path and degree-constraint information, each node eliminates unsuitable alternate paths. It returns the shortest alternate path found.

Heuristic Fixup's query to each of the over-constrained node's tree neighbors is passed on to their neighbors and further in the network until the message reaches the farthest nodes in the neighborhood. Each node then returns the shortest alternate path found by either itself or its neighbors. Of the alternate paths reported, heuristic Fixup picks the best one. It then deletes the now redundant over-constrained edge and replaces it with the alternate path. It does so by commanding the head of the alternate path to reconnect the tree. In addition, it sends a disconnect message to its former neighbor at the other end of the over-

```
for all neighbors do
      send request for alternate paths to neighbor
wait for responses

if shortest alternate path exists then
      send connect message to head of alternate path
wait for connect status

if success then
      drop redundant adjacent edge
      pass control to closest over-constrained node
else
      # connect failure, heuristic fails
```

Figure 14: Pseudocode for over-constrained node in heuristic Fixup.

constrained edge. If this former neighbor is now a non-multicast leaf node, it deletes itself and its edge, forwarding the disconnect message to *its* former neighbor. This process repeats until all non-multicast leafs created are deleted. Heuristic Fixup's pseudocode is shown in Figure 14.

Messages and Convergence Time: In the best case, Heuristic Fixup uses the least number of messages when the initial tree is already degree constrained. Since the leader node has the complete tree's topology and degree constraints, this case requires zero messages and no convergence time. In our test networks, 6.7% of the solutions produced by unconstrained distributed K-SPH satisfied the degree constraints without post-processing. Likewise, 8.5% of the solutions produced by unconstrained distributed SPH also did not require post-processing.

In the worst case for degree-constrained distributed heuristic Fixup, every node in the tree is over-constrained. This forces heuristic Fixup to run on all Z nodes of the tree. Each node queries its neighbors for alternate paths. The number of such messages is the average degree raised to the neighborhood size. Thus, the number of messages $= O(Z)$. For example, our test networks have an average degree of three and we employ a neighborhood of three hops which leads to the constant $= 3^3 = 27$. Likewise, the worst case convergence time occurs when each message in the tree must travel a distance at most twice the diameter of the graph, $2D$. Thus, the convergence bound for heuristics Fixup $= O(DZ)$.

The convergence-time and message bounds for the three distributed heuristics are summarized in Tables 1 and 2, respectively.

4 Simulation Results

To compare the three DCSP heuristics presented in the last section, we implemented the algorithms and performed extensive simulations. This section summarizes the simulation results and compares the algorithms in terms of their convergence time, competitiveness, and percentage of unsolved cases.

Bound	Distributed K-SPH	Distributed SPH	Distributed Fixup
Lower Bound	$\log Z$	DZ	0
Upper Bound	DZ	DZ	N.A.
Degree-Constrained Upper Bound	DNZ	DNZ	DZ

Table 1: Convergence-time bounds for distributed heuristics K-SPH, SPH and Fixup.

Bound	Distributed K-SPH	Distributed SPH	Distributed Fixup
Lower Bound	$Z \log Z$	Z^2	0
Upper Bound	ZN	ZN	N.A.
Degree-Constrained Upper Bound	ZN^2	ZN^2	Z

Table 2: Messages bounds for distributed heuristics K-SPH, SPH and Fixup.

4.1 Evaluation Methodology

4.1.1 Network model

Because our choice of existing network topologies and multicast applications was small, we chose to compare DCSP heuristics using randomly generated networks. Each algorithm was run on a total of 2000 test networks. Each of the 2000 networks is a sparse 200-node network with a degree-constraint on 50 or 75% of its nodes. We consider an n-node graph to be sparse when less than 5% of the possible $\binom{n}{2}$ edges are present in the graph. For our 200-node test networks, this means the number of edges must fall in the narrow range between the minimum number of edges for a 200-node graph and $\binom{n}{2}$ edges out of a possible 19,900 edges for a complete graph: 199 — 995 edges. Figure 15 shows the distribution of number of edges for our test networks. We believe these networks describe plausible networks of point-to-point nodes in a WAN because such large networks are likely to be loosely interconnected and may have many degree-constrained nodes. Likewise, the simulated networks have 10% or 30% of its nodes in the multicast group because multicast applications running on such a WAN are likely to involve only a minority of nodes in the network.

The 2000 networks were generated to resemble networks in a manner similar to that of Doar [11]. Each of the n nodes is distributed across a Cartesian coordinate plane with minimum and maximum coordinates $(0, 0)$ and $(2n, 2n)$, creating a forest of n nodes spread across this plane. In all of our test graphs, the number of nodes n was set at 200. The nodes are then connected by a random spanning tree. This tree is generated by iteratively considering a random edge between nodes and accepting those edges that connect distinct components. The remaining redundant edges of the graph are chosen by examining each possible edge (x, y) and generating a random number $0 \leq r < 1$. If r is less than a probability function $P(x, y)$ based on the distance between x and y, then the edge is accepted. Each edge's distance is its rectilinear distance plus a small constant. This distance is also the time it takes for a message to traverse this edge. We used

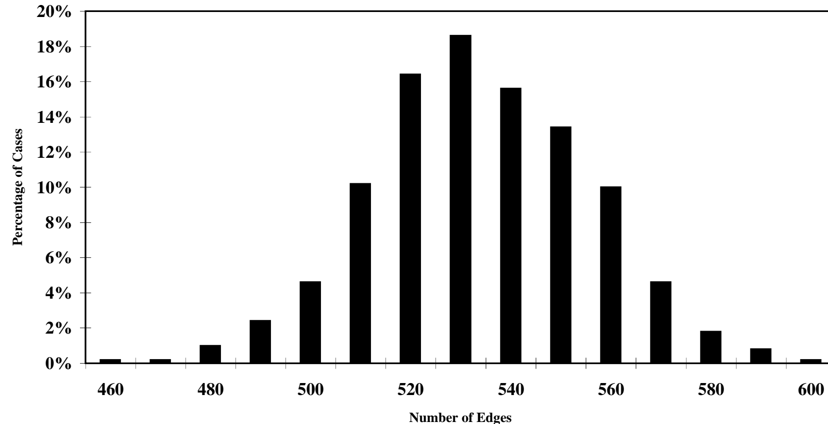

Figure 15: The histogram of the number of edges in the test graphs.

the probability function

$$P(x,y) = \beta e^{\frac{-d_{x,y}}{2\alpha n}},$$

where $d_{x,y}$ is the rectilinear distance between nodes x and y [11]. The parameters α and β govern the density of the graph. Increasing α increases the number of connections to nodes far away and increasing β increases the number of edges from each node. After some experimentation, we chose $\alpha = 0.10$ and $\beta = 0.20$ for generating the graphs used in this simulation. These values produced graphs of realistic density and degree-distribution.

We performed four different simulations on each generated graph by varying the multicast group size and the percentage of degree-constrained nodes, each in two ways. The number of multicast nodes was chosen as either 20 or 60 of the 200 nodes; the number of degree-constrained nodes was selected as either 100 or 150. Results are presented for all four combinations for each graph. Degree-constrained nodes in each case were chosen randomly and each node's degree-constraint was chosen randomly between 2 (no multicast capability) and one less than the degree of the node. Similarly the nodes in a multicast group were chosen randomly in each case. The random numbers were chosen from a uniform distribution. Figure 16 shows the distribution of degree-constraints for our test networks. Nearly a fifth of the nodes have degree constraints two — no multicast capability — because so many of the graph nodes have low degree. In addition to our random degree-constraints, we also simulated our distributed heuristics on 1000 test networks with a degree constraint of three on all nodes. These more difficult test networks serve two purposes (i) they further test our degree-constrained heuristics under more stringent conditions and (ii) they represent an environment in which degree-constraints are generally much less than than the degree of a switch.

To ensure fairness, each heuristic was run on the same networks. Their results are discussed in Section 4.3.

4.2 The Heuristic Simulator

Each of the heuristics was implemented on top of our degree-constrained Steiner problem simulation platform, designed to provide the level playing field upon which to base comparisons. It supplies the basic graph

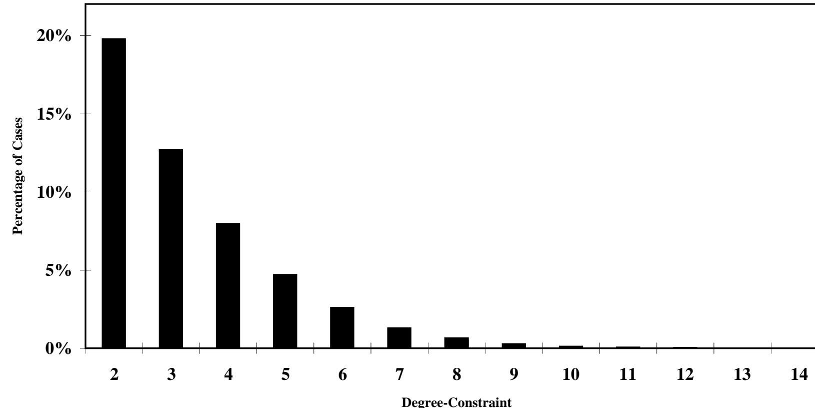

Figure 16: The histogram of degree-constraints for our test networks.

manipulation routines used by the heuristics such as adding and deleting edges.

The metrics we use for comparison are the competitiveness, convergence time, and percentage of cases where the algorithm succeeded in constructing a multicast tree satisfying the given degree constraints. Competitiveness is defined to be the ratio between the cost of the solution tree (sum of edge distances) produced by the algorithm under consideration to the cost of the best solution found by any of the evaluated heuristics, including both centralized and distributed versions, for that test case. Convergence time is expressed in terms of thousands of time units, where each time unit is defined as the time for a message to propagate in a link of unit distance.

4.3 Results

Having described the heuristics evaluated and the simulation environment, we turn now to the results of our simulations.

Figure 17 displays the distribution of competitiveness for centralized SPH and K-SPH, distributed SPH and K-SPH, and heuristic Fixup. Each of the five curves shows the cumulative percentage of cases whose competitiveness is less than or equal to a given value. Distributed K-SPH was used to generate the initial unconstrained solutions for the Fixup heuristic.

The competitiveness for most cases shown in Figure 17 fell well within 10% of the best heuristic solution found. Even the worst-performing heuristic, heuristic Fixup, solved the majority of its test cases within 5% of the best solution found. This high competitiveness for all evaluated heuristics is similar to our findings for centralized Steiner heuristics and shows that factors other than the competitiveness may determine the choice of the algorithm among the three.

The distributed versions of SPH and K-SPH performed slightly inferior in competitiveness as compared to their centralized versions; this is because of the lack of global network topology information that is available in the centralized versions. However, the differences in quality were not significant. In fact, distributed K-SPH often outperformed centralized SPH. Among the three heuristics evaluated, heuristic K-SPH consistently outperformed both heuristic SPH and heuristic Fixup.

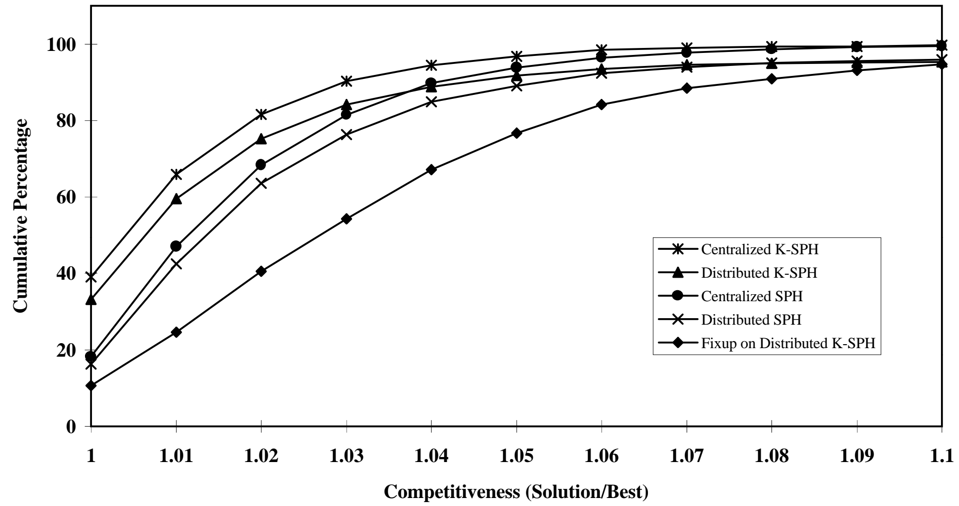

Figure 17: The cumulative competitiveness histogram for degree-constrained graphs.

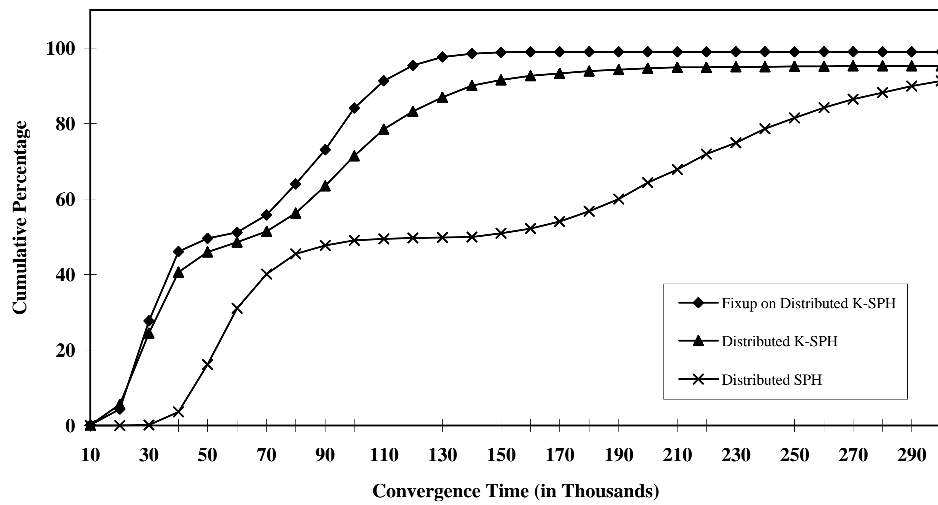

Figure 18: The cumulative convergence time histogram for degree-constrained graphs.

Heuristic	Percentage
Fixup on distributed K-SPH	99.0%
Distributed SPH	96.4%
Distributed K-SPH	95.6%

Table 3: Percentage of cases solved using distributed DCSP heuristics.

Heuristic Fixup, however, does enjoy an advantage over distributed heuristics SPH and K-SPH in convergence time as shown by Figure 18. This figure displays the cumulative percentage of networks solved within a given convergence time. Note that since our simulations are split between cases with 20 multicast members and those with 60 multicast members, the plots in the figure exhibit two peaks: one for 20 multicast members and one for 60 multicast members. Convergence time for heuristic Fixup is the sum of heuristic K-SPH's convergence time and the convergence time of heuristic Fixup. Since heuristic K-SPH converges faster when run on an unconstrained graph than on a degree-constrained one and since heuristic Fixup converges in a fraction of K-SPH's convergence time (see Figure 20), the total convergence time for this combination edges out distributed K-SPH run on degree-constrained graphs. This result conforms to the asymptotic bounds given in Table 1. Both distributed heuristics SPH and K-SPH have a higher upper bound for degree-constrained graphs because of the additional complexity of finding degree-constrained paths. This additional complexity is also reflected in the bounds for messages as shown by Table 2.

Comparing the two distributed heuristics SPH and K-SPH, heuristic K-SPH's convergence time is superior to that of SPH. The reason that distributed K-SPH converges more quickly than distributed SPH is that the size of fragments in the K-SPH algorithm starts small (one node per fragment) and tends to remain small. If every fragment in distributed K-SPH always found a partner, the final two fragments would each contain about half the nodes in the final tree. Distributed heuristic SPH, by contrast, starts with a single fragment of one node that grows at most a few multicast members at a time until it contains all the multicast members. With increasing fragment size comes increasing coordination overhead between tree nodes. This increased overhead is evident in distributed SPH's elongated plot for 60 multicast members in Figure 18.

Table 4 summarizes the percentage of test networks solved using the distributed heuristics. It is remarkable that all three heuristics solved greater than 95% of the test networks. Among the heuristics, heuristic Fixup solved the largest number of networks. Distributed heuristic SPH was next and was better than distributed K-SPH by less than one percent for the 2000 test networks. We believe that the larger amount of topology information available to the single, large fragment in distributed heuristic SPH allowed it to solve a slightly larger number of our test networks. Similarly, distributed heuristic Fixup has fairly complete topology information about the test network before even beginning the heuristic and this allows Fixup to make better choices when building a degree-constrained tree.

Heuristic Fixup's results depend not only on the initial tree, but also on how far we allow Fixup to search for alternate paths to over-constrained nodes. To determine the effect of the size of the neighborhood examined by the heuristic for alternate solutions on its performance, we ran the heuristic with neighborhood sizes of 2, 3, 4, and 6 hops. The results are summarized in Figures 19 and 20 as well as in Table 4. A neighborhood size of 3 to 6 hops allowed heuristic Fixup to solve at least 99% of our test networks. However, reducing the neighborhood size to 2 drastically reduced the number of test networks solved to 86.1%. This is because few good alternate paths were available very close to the nodes violating degree constraints.

Hops	Percentage
6	99.8%
4	99.0%
3	99.0%
2	86.1%

Table 4: Percentage of cases solved by distributed K-SPH + Fixup using 2, 3, 4 and 6 hops.

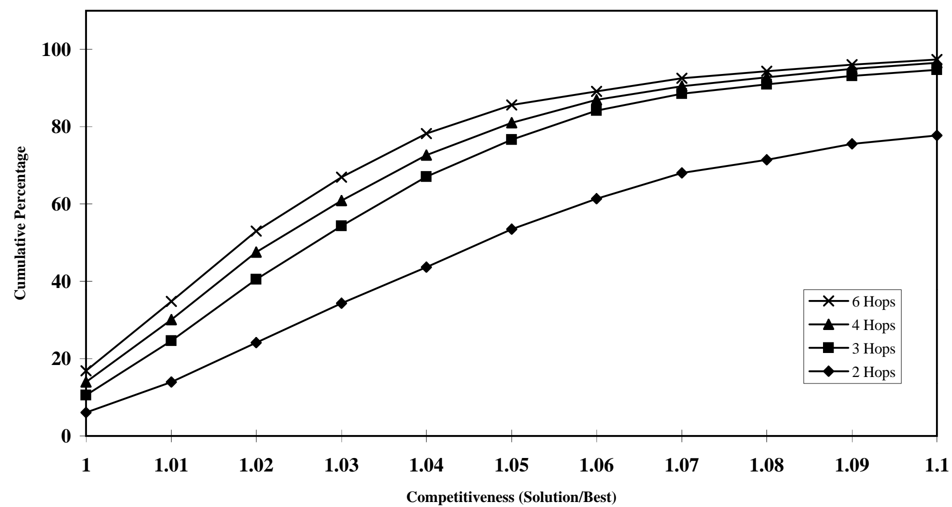

Figure 19: Heuristic Fixup's cumulative competitiveness histogram for 2, 3, 4, and 6 hops.

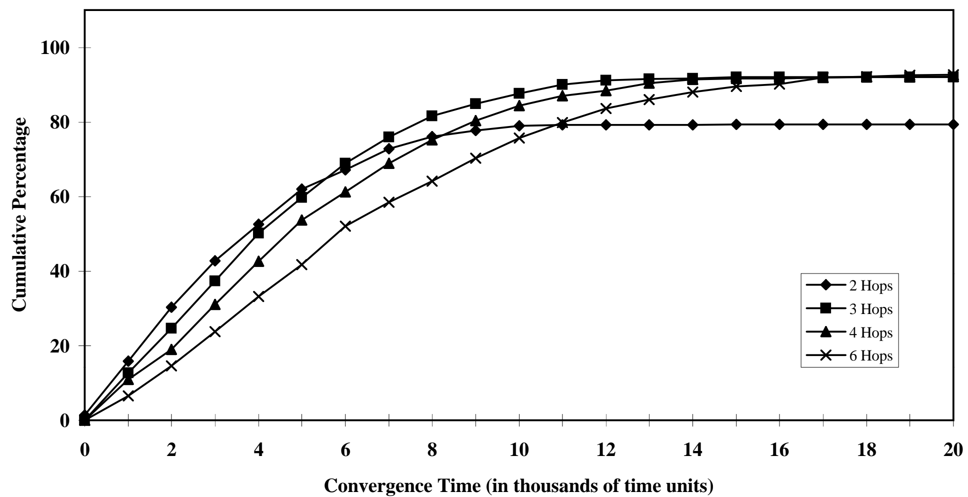

Figure 20: Heuristic Fixup's cumulative convergence time histogram for 2, 3, and 4, and 6 hops.

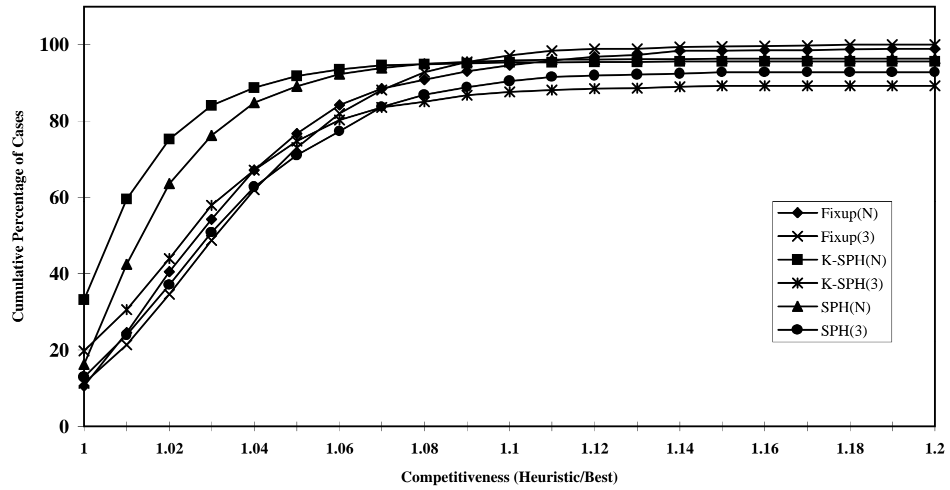

Figure 21: The cumulative competitiveness histogram for graphs with a degree-constraint of 3.

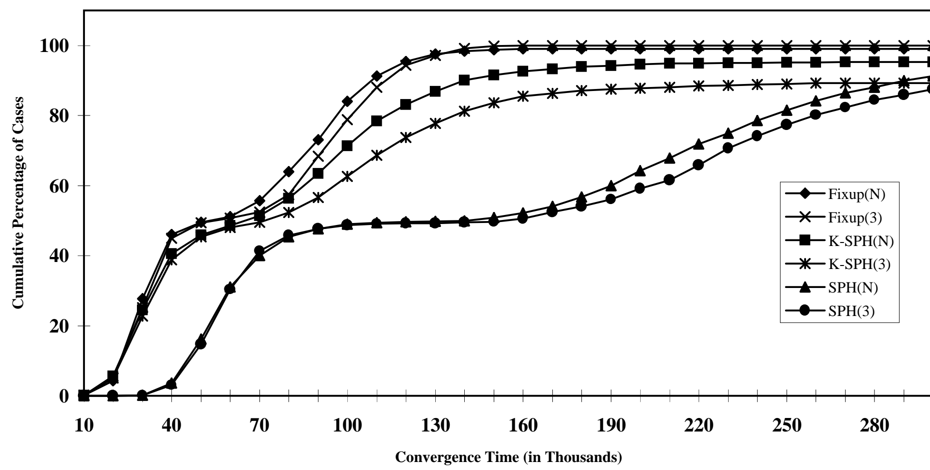

Figure 22: The cumulative convergence time histogram for graphs with a degree-constraint of 3.

Heuristic	Percentage
Fixup on distributed K-SPH	100.0%
Distributed SPH	92.8%
Distributed K-SPH	89.3%

Table 5: Percentage of cases solved using distributed DCSP heuristics when degree-constraint set at 3.

Similarly, degree-constrained trees built using a neighborhood size of 2 hops were inferior in quality to those built using neighborhoods of 3, 4 and 6. The differences, however, between trees built using neighborhoods 3, 4 and 6 hops were slight.

Finally, the convergence time for heuristic Fixup was best overall for a neighborhood size of 3 hops. With 2 hops, the convergence time was promising for the easiest cases, but became far too high for the remaining, harder cases.

Although setting the neighborhood size to 6 hops allows heuristic Fixup to solve almost all of our test networks, a neighborhood size of 3 represents the best tradeoff between number of cases solved, competitiveness, and convergence time for our test networks. Hence, this was the neighborhood size used in the results of Figure 17 and 18. This best neighborhood size may vary for other networks.

For example, in the more stringent degree-constrained environment of Figures 21 and 22, fixup was able to solve only 69.3% of the cases when its neighborhood was set to 3. In these simulations, the degree-constraint of all nodes is set to 3. When Fixup's neighborhood is doubled to 6 hops, the number of cases solved jumps to 100% as shown in Table 5. Ironically, heuristic Fixup is even able to solve the one case out of 1000 that neither distributed K-SPH nor SPH could solve. In these figures competitiveness and convergence time are shown in relation to those cases where degree-constraints are randomly distributed. Each heuristic in both figures either has "(3)" appended to its name meaning that node degree-constraints are fixed at 3 or "(N)" appended to its name meaning that node degree-constraints are variable. For all three heuristics, the fixed degree-constraint cases performs slightly worse relative to the random degree-constrained case.

Even though the competitiveness of all three heuristics with a fixed degree-constraint of 3 trail their random degree-constraint equivalents, they continue to produce the majority of solutions well within 10% of the best solution found. Similarly, convergence time for all three heuristics is longer, but not significantly. From these results we draw the conclusion that the degree-constrained heuristics SPH and K-SPH are relatively insensitive to degree-constraints, degrading slowly when node degree-constraints becomes tighter. Heuristic Fixup, however, is sensitive to the severity of degree-constraints and needs an increased neighborhood size to compensate for tighter degree-constraints.

5 Concluding Remarks

In this paper we studied the degree-constrained multicast tree problem as applied to point-to-point networks and evaluated the effectiveness of two distributed heuristics and one post-processing heuristic. Each of the evaluated heuristics enjoys the advantage that it does not require the nodes to have knowledge of the complete topology of the network. Instead, local information about neighbors is sufficient. In addition, only those nodes directly involved in the multicast tree participate, unlike spanning-tree based heuristics. These heuristics were compared in terms of their competitiveness, convergence time, and the number of networks

they could not solve. Surprisingly few (less than 5%) of our test networks were unsolvable for random degree-constraints. With a constant degree-constraint of 3, the number of unsolved cases rose to 10% for heuristics SPH and K-SPH. By allowing heuristic Fixup a larger neighborhood size, its number of unsolved cases remained the same.

Perhaps the most interesting result from our simulations is that while the degree-constrained distributed heuristics compare favorably with their centralized versions and solved most of the test networks, post-processing unconstrained Steiner trees often produce better solutions. Post-processed Steiner trees lost little of their quality and were within 10% of the best solution found by any heuristic — centralized, or distributed.

While the focus of this paper is on distributed Steiner heuristics to find multicast trees in data networks with degree-constraints, our initial results show that the same heuristics produce good quality solutions in the general, unconstrained problem. In particular, their competitiveness is often superior to that produced by previous heuristics [8, 22] that are based on a distributed minimum spanning tree algorithm. A detailed comparison of the performance of these distributed heuristics on unconstrained networks will appear in [2].

References

[1] F. Bauer and A. Varma. "Degree-constrained multicasting in point-to-point networks," in *Proc. IEEE INFOCOM*, Boston, Apr. 1995, pp. 369–376.

[2] F. Bauer and A. Varma. "Distributed algorithms for multicast path setup in data networks," in *Proc. IEEE GLOBECOM*, Singapore, Nov. 1995.

[3] J. Beasley. "An SST-based algorithm for the Steiner problem in graphs," *Networks*, vol. 19, pp. 1–16, 1989.

[4] L. Berry. "Graph theoretic models for multicast communications," in *Traffic theories for new telecommunications services ITC Specialists Seminar*, Adelaide, Australia, Sep. 1989, pp. 95–99.

[5] K. Bharath-Kumar and Jaffe. "Routing to multiple destinations in computer networks," *IEEE Transactions on Communications*, vol. COM-31, no. 3, pp. 343–351, Mar. 1983.

[6] J. Byun and T. Lee. "The design and analysis of an ATM multicast switch with adaptive traffic controller," *IEEE Transactions on Networking*, vol. 2, no. 3, pp. 288–298, Jun. 1994.

[7] H. Chao and B. Choe. "Design and analysis of a large-scale multicast output buffered ATM switch," *IEEE Transactions on Networking*, vol. 3, no. 2, pp. 126–138, Apr. 1995.

[8] G. Chen, M. Houle, and M. Kuo. "The Steiner problem in distributed computing systems," *Information Sciences*, vol. 74, no. 1-2, pp. 73–96, Oct. 1993.

[9] C. Cheng, R. Riley, S. Kumar, and JJ. Garcia-Luna-Aceves. "A loop-free extended Bellman-Ford routing protocol without bouncing effect," *ACM Computer Communications Review*, vol. 19, no. 4, pp. 224–236, Sep. 1989.

[10] S. Deering. "Multicast routing in internetworks and extended LANs," *Computer Communication Review*, vol. 18, no. 4, pp. 55–64, Aug. 1988.

[11] M. Doar and I. Leslie. "How bad is naive multicast routing?," in *Proc. IEEE INFOCOM*, San Francisco, CA, Apr. 1993, pp. 82–89.

[12] R. Douglas. "NP-completeness and degree restricted spanning trees," *Discrete Mathematics*, vol. 105, pp. 41–47, 1992.

[13] R. Gallager, P. Humblet, and P. Spira. "A distributed algorithm for minimum-weight spanning trees," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 1, pp. 66–77, Jan. 1983.

[14] P. Humblet. "Another adaptive distributed shortest path algorithm," *IEEE/ACM Transactions on Communications*, vol. 39, no. 6, pp. 995–1003, Jun. 1991.

[15] F. Hwang and D. Richards. "Steiner tree problems," *Networks*, vol. 22, pp. 55–89, 1992.

[16] F. Hwang, D. Richards, and P. Winter. *The Steiner Tree Problem.* New York: North-Holland, 1992.

[17] X. Jiang. "Distributed path finding algorithm for stream multicast," *Computer Communications*, vol. 16, no. 12, pp. 767–775, Dec. 1993.

[18] D. Johnson. "The NP-completeness column: an ongoing guide," *Journal of Algorithms*, vol. 6, no. 3, pp. 434–451, Sep. 1985.

[19] H. Kim. "Design and performance of multinet switch: a multistage ATM switch architecture with partially shared buffers," *IEEE Transactions on Networking*, vol. 2, no. 6, pp. 581–587, Dec. 1994.

[20] V. Kompella, J. Pasquale, and G. Polyzos. "Multicasting for multimedia applications," in *Proc. IEEE INFOCOM*, New York, NY, May 1992, pp. 2078–2085.

[21] V. Kompella, J. Pasquale, and G. Polyzos. "Multicast routing for multimedia communications," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 286–292, Jun. 1993.

[22] V. Kompella, J. Pasquale, and G. Polyzos. "Two distributed algorithms for the constrained Steiner tree problem," in *Proc. Comput. Commun. and Netw.*, San Diego, CA, Jun. 1993.

[23] J. Kruskal. "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. Amer. Math. Soc.*, vol. 7, pp. 48–50, 1956.

[24] J. Rugelj. "Distributed multicast routing mechanism for global point-to-point networks," in *Proceedings of the 20th EUROMICRO Conference*, Liverpool, UK, Sep. 1994, pp. 389–395.

[25] M. Smith and P. Winter. "Path-distance heuristics for the Steiner problem in undirected networks," *Algorithmica*, vol. 7, no. 2-3, pp. 309–327, 1992.

[26] H. Takahashi and A. Matsuyama. "An approximate solution for the Steiner problem in graphs," *Math. Japonica*, vol. 24, no. 6, pp. 573–577, 1980.

[27] H. Tode, Y. Sakai, M. Yamamoto, H. Okada, and Y. Tezuka. "Multicast routing algorithm for nodal load balancing," in *Proc. IEEE INFOCOM*, New York, NY, May 1992, pp. 2086–2095.

[28] J. Turner. "An optimal nonblocking multicast virtual circuit switch," in *Proc. IEEE INFOCOM*, Toronto, Canada, Jun. 1994, pp. 298–305.

[29] S. Voss. "A survey of some generalizations of Steiner's problem," in *Proc. of the First Balkan Conference on Operational Research*, 1988.

[30] S. Voss. *Steiner-Probleme in Graphen.* Frankfurt/Main: Hain, 1990, pp. 179–184.

[31] S. Voss. "Problems with generalized Steiner problems," *Algorithmica*, vol. 7, no. 2-3, pp. 333–335, 1992.

[32] S. Voss. "Steiner's problem in graphs: Heuristic methods," *Discrete Applied Mathematics*, vol. 40, pp. 45–72, 1992.

[33] P. Winter. "Steiner problem in networks: A survey," *Networks*, vol. 17, no. 2, pp. 129–167, 1987.

[34] W. De Zhong, Y. Onozato, and J. Kaniyil. "A copy network with shared buffers for large-scale multicast ATM switching," *IEEE/ACM Transactions on Networking*, vol. 1, no. 2, pp. 157–165, Apr. 1993.