INFORMS Journal on Computing
Vol. 17, No. 3, Summer 2005, pp. 290–304
ISSN 1091-9856 | EISSN 1526-5528 | 05 | 1703 | 0290

DOI 10.1287/ijoc.1040.0077

Heuristic Search for the Generalized Minimum Spanning Tree Problem

Bruce Golden, S. Raghavan, Daliborka Stanojević
The Robert H. Smith School of Business, University of Maryland, College Park, Maryland 20742-1815, USA
{bgolden@rhsmith.umd.edu, raghavan@umd.edu, dstanoje@rhsmith.umd.edu}

The generalized minimum spanning tree (GMST) problem occurs in telecommunications network planning, where a network of node clusters needs to be connected via a tree architecture using exactly one node per cluster. The problem is known to be NP-hard, and even finding a constant factor approximation algorithm is NP-hard. In this paper, we present two heuristic search approaches for the GMST problem: local search and a genetic algorithm. Our computational experiments show that these heuristics rapidly provide high-quality solutions for the GMST and outperform some previously suggested heuristics for the problem. In our computational tests on 211 test problems (including 169 problems from the TSPLIB set), our local-search heuristic found the optimal solution in 179 instances and our genetic-algorithm procedure found the optimal solution in 185 instances (out of the 211 instances, the optimal solution is known in 187 instances). Further, on each of the 19 unsolved instances from TSPLIB, both our local-search heuristic and genetic-algorithm procedure improved upon the best previously known solution.

Key words: networks; tree algorithms; heuristics; local search; genetic algorithms
History: Accepted by Michel Gendreau, Area Editor for Heuristic Search and Learning; received January 2003; revised October 2003; accepted February 2004.

1. Introduction

The generalized minimum spanning tree (GMST) problem is an important network design problem that arises within the scope of telecommunications network planning. In this problem setting, we are given an undirected graph $G = (V, E)$, with node set V and edge set E, and a cost vector $\mathbf{c} \in \mathcal{R}_+^{|E|}$ on the edges E. We are also given a set of K mutually exclusive and exhaustive node sets $V_1, \ldots, V_K$ (i.e., $V_i \cap V_j = \phi$, if $i \neq j$, and $\bigcup_{k=1}^{K} V_k = V$). We wish to find a minimum-cost tree that contains exactly one node from each cluster. Figure 1 gives an example of a generalized spanning tree for a network with five clusters. One application of the GMST problem, first introduced in Myung et al. (1995), is in the regional connection of local area networks (LANs). In this application, several LANs in a region need to be connected with each other. For this purpose, one gateway node needs to be identified within each LAN, and the gateway nodes are to be connected via a minimum-cost spanning tree (MST).

The GMST problem has been studied by only a few researchers. Myung et al. (1995) showed that the problem is NP-hard, and even finding a constant factor approximation algorithm is NP-hard. They also developed a dual-ascent procedure based on a multicommodity flow formulation for the problem and used this within a branch-and-bound framework to solve problems with up to 100 nodes and 4,500 edges to optimality. Feremans et al. (2002) described eight different integer programming (IP) and mixed-integer programming (MIP) formulations for the GMST problem and show that four of these formulations strictly dominate the other four, in terms of the quality of their linear relaxations. Recently, Raghavan (2002) showed that the GMST may be modeled as a Steiner tree problem with degree constraints on some of the nodes. He showed that the resulting formulation is equivalent in strength (in terms of the linear relaxation) to the four tightest formulations identified in Feremans et al. (2002).

Feremans (2001) presented several new classes of valid inequalities and developed a branch-and-cut algorithm for the GMST problem. She reported success in solving, to optimality, instances where the cost function satisfies the triangle inequality with up to 160 nodes in less than two hours of CPU time (these instances are much more difficult than random instances), and random instances (i.e., where the cost function does not satisfy the triangle inequality) with up to 200 nodes in less than two hours of CPU time. As part of the branch-and-cut procedure, Feremans (2001) employed a tabu-search heuristic to generate an initial upper bound, and a local-search procedure to improve upper bounds found during the course of the branch-and-cut procedure. A subset of these results is included in Feremans et al. (2004). However,

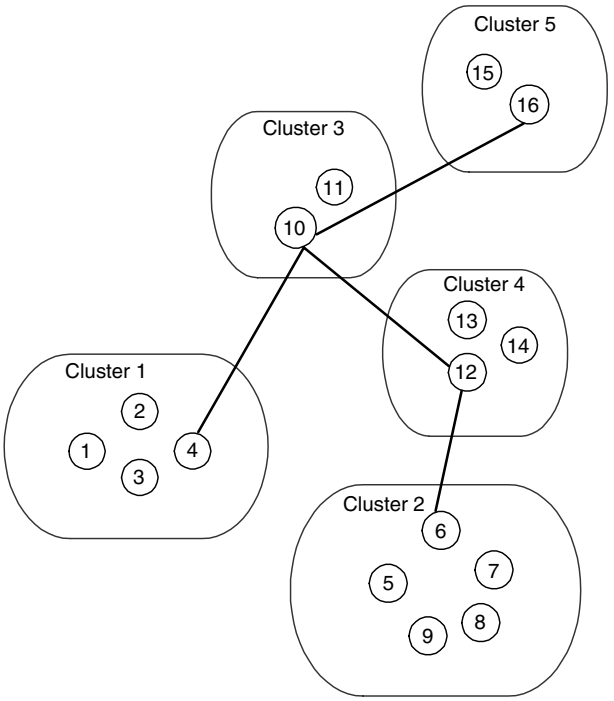

Figure 1 **An Example of a Generalized Spanning Tree for a Network with Five Clusters**

separate computational results of their tabu-search and local-search heuristics, as stand-alone procedures, are not reported.

Pop et al. (2000) provided a polynomial-size MIP formulation for the GMST problem. They described a relaxation procedure that seems to provide strong lower bounds in their preliminary computations. Although the GMST is inapproximable to a constant factor, unless P = NP, Pop et al. (2001) described a very interesting result. They showed that when the cluster size is bounded (say by ρ), then it is possible to provide a constant factor approximation; they described a 2ρ-approximation algorithm for the GMST with bounded cluster size.

Dror et al. (2000) studied a somewhat different variation of the GMST. In their problem, the clusters need not be mutually exclusive (but are collectively exhaustive). Further, instead of having exactly one node from a cluster in the desired tree, they require that at least one node from each cluster be in the tree (thus, allowing multiple nodes from the same cluster to be in the tree). Dror et al. (2000) described several simple heuristics along with a genetic algorithm for their problem. In all the examples considered in their paper, the genetic algorithm provided better solutions than did the simple heuristics. Feremans (2001) compared the genetic-algorithm results against the optimal solution, and found that the genetic-algorithm solutions are, on average, 6.53% from optimality.

In this paper, we describe two heuristic search procedures for the GMST problem. First, we describe a local-search heuristic for the GMST problem. We then describe a more sophisticated genetic algorithm that employs four different genetic operators—two reproductive and two mutative. We test these heuristics on two sets of large problem instances (instances where the cost satisfies the triangle inequality and random instances) and find that the local-search heuristic generates solutions that are, on average, 1.26% from optimality. The genetic algorithm generates solutions that are, on average, 1.18% from optimality. On closer examination, of the 211 problem instances, our local-search heuristic took an average of 23.1 seconds and found the optimal solution in 179 out of 187 instances for which the optimal solution is known. Our genetic-algorithm procedure took an average of 40.7 seconds and found the optimal solution in 185 out of 187 instances for which the optimal solution is known. Over the 187 instances for which the optimal solution is known, the local-search heuristic generates solutions that are, on average, 0.07% from optimality, while the genetic-algorithm procedure generates solutions that are, on average, 0.01% from optimality. Thus, the value of the gaps (over all the 211 instances) is exaggerated by the fact that, for many of the problem instances where the optimal solution is unknown, the quality of the lower bound is poor.

The remainder of this paper is organized as follows. In §2, we present simple lower and upper bounding heuristics. The lower bound could be useful when it is not possible to obtain lower bounds from the relaxations of the MIP formulations for the GMST problem in a reasonable amount of time. The upper bounding procedure could be used to generate initial solutions for a local-search procedure, or as an upper bound for a branch-and-bound algorithm for the problem. The upper bounding procedures are adaptations of Prim's, Kruskal's, and Sollin's algorithms for the MST (see the text by Ahuja et al. (1993) for a nice description of those algorithms). However, there are some interesting choices that need to be made in the adaptation of these algorithms to the GMST that we explore within this paper. In §3, we describe our local-search procedure. We then develop our genetic algorithm for the GMST problem in §4. Finally, in §5, we report on our computational experience with the two heuristic search procedures. In §6, we provide some concluding remarks.

2. Simple Lower and Upper Bounds for the GMST

In this section, we describe simple lower and upper bounding procedures for the GMST.

2.1. Spanning Tree Lower Bound

A lower bound for the GMST may be obtained by solving the minimum spanning tree problem on the following transformed graph H. Replace each clus-

ter by a single node. The cost of an edge between two nodes in the transformed graph is equal to the minimum-cost edge between the two clusters that the two nodes represent. To calculate this lower bound, it is not necessary to contract the graph. Instead, it is fairly easy to modify the greedy algorithm for the MST, to accomplish the same. For completeness, we describe the steps below.

Spanning Tree Lower Bound:

Step 1. Start with a completely disconnected graph T with K clusters defined.

Step 2. Order the edges of the initial graph G in ascending order of their cost.

Step 3. Starting from the beginning of the list, add edges to T provided that this addition does not create a cycle among the clusters, when each cluster is contracted (in concept only) to a single node.

Step 4. Repeat Step 3 until $K-1$ edges have been added.

The cycle condition in Step 3 above can be easily checked in a typical implementation of Kruskal's algorithm by making the following minor modification. Recall, in Kruskal's algorithm, a linked list is used to represent each forest in the network constructed at any stage of the algorithm. In the initialization of Kruskal's, each node is represented as a linked list with a single item. To calculate the lower bound without contracting the graph, we initialize the algorithm with each cluster represented as a linked list containing the items in the cluster. Consequently, the running time of the spanning tree lower bound procedure is identical to Kruskal's and is $\mathcal{O}(|E|+|V|\log|V|)$. We note that a slight speedup in Step 2 may be obtained by noting that only the minimum-cost edge between two clusters can be added in Step 3. Thus, by spending an additional $\mathcal{O}(|E|)$ time, we can determine the edges that actually need to be sorted.

2.2. Upper Bound Procedures

As noted by Feremans (2001) and Feremans et al. (2004), an upper bound may be easily obtained by straightforwardly adapting Kruskal's, Prim's, or Sollin's algorithm for the MST to the GMST. However, in an adaptation of Prim's or Sollin's algorithm, there are several choices that could be made, which affect the quality of the results. In this section, we examine the effect of these choices to develop an improved upper bound heuristic.

To elaborate, we briefly review how Kruskal's, Prim's, and Sollin's algorithms operate. Kruskal's algorithm for the MST adds edges greedily (until a spanning tree is obtained) unless the addition of an edge causes a cycle. This may be adapted in a straightforward fashion to the GMST as follows. Add edges in ascending order, unless the edge creates a cycle or is incident to a second node in any cluster. The running time remains the same as Kruskal's for the MST and so is $\mathcal{O}(|E|+|V|\log|V|)$ time.

Prim's algorithm for the MST starts from a node and grows the MST. It does this by considering the partially constructed tree and adding the minimum-cost edge from nodes in the tree to nodes not in the partially constructed tree. Thus, the very first step in an adaptation of Prim's algorithm to the GMST requires the choice of a starting node (the choice of the starting node can result in different solutions for the GMST). Next, for the partially constructed tree, we add the minimum-cost edge from nodes within the tree to nodes that are in clusters distinct from the nodes in the partially constructed tree. Since we need to select a starting node for Prim's adaptation to the GMST, one possibility is to select a starting node randomly. A second possibility is to try all $|V|$ nodes as starting nodes and output the best solution. The running time for Prim's adaptation is the same as Prim's for the MST, so is $\mathcal{O}(|E|+|V|\log|V|)$. (Actually, the running time is slightly faster. The number of nodes and edges in the running-time equation can be reduced to the number of nodes and edges in the graph obtained by deleting all nodes, other than the starting node, in the starting node's cluster.)

Sollin's algorithm for the MST starts with each node representing a tree. In each iteration of Sollin's algorithm, it identifies, for each tree in the partial solution, the minimum-cost edge emanating from the tree. It then adds these edges to the partial solution (thus merging trees to build larger trees, and reducing the number of trees in the partial solution). The iterations of the algorithm are repeated until a spanning tree is obtained. We adapt Sollin's algorithm to the GMST as follows. We first sort the edges of the graph into increasing order. In each iteration of the algorithm, for each tree (or cluster, if no edge in the forest constructed so far is incident to any node in the cluster) select as a candidate edge the minimum-cost edge out of the tree (or cluster) whose addition is feasible (i.e., adding the edge will not result in multiple nodes from a cluster in the partial solution). Ties between edges, for selection as candidate edge, are broken by choosing the edge that appears first in the sorted order. Consider the selected edges in sorted order and add an edge to the partial solution if its addition is feasible. (Note that, although the edges were feasible when selected, once we start adding edges to the partial solution, a selected edge may no longer be feasible for addition to the partial solution.) We repeat these iterations until a feasible generalized spanning tree (GST) is obtained. Each iteration of this adaptation takes $\mathcal{O}(|E|+K)$ time as we go through the list of $|E|$ sorted edges to select the minimum-cost feasible edge out of each tree (or cluster) and then consider at most $K-1$

edges to add in an iteration. In each iteration at least one edge is added since it is always feasible to add the first selected edge. Thus there are at most $K-1$ iterations. Consequently, the overall running time is $\mathcal{O}(|E|K)$ plus the time for initially sorting the edges.

Observe that, with Kruskal's and Sollin's adaptation, the heuristic solution is unique, while with Prim's the solution is dependent on the starting node selected. It is also easy to establish that the tree generated by each of these algorithms is an MST on the nodes in the tree. We conducted some preliminary experiments to compare the quality of the solutions produced by the three adaptations. Specifically, we compared Kruskal's adaptation, Prim's adaptation with a random choice of starting node, and Sollin's adaptation. We found that Kruskal's and Sollin's algorithm outperformed Prim's algorithm. To illustrate the differences, in Table 1 we compare the three algorithms on TSPLIB instances from §5, where the optimal solution is known. We find that, with each of the different clustering types (§5 provides further details on the dataset and clustering types), on average, Kruskal's and Sollin's outperform Prim's. Further, as the average number of nodes in a cluster increases (μ may be thought of as representing the average number of nodes in a cluster), the quality of Kruskal's, Prim's, and Sollin's upper bounds deteriorate.

In Table 1, we also evaluate the quality of the spanning tree lower bound. We found that the spanning tree lower bound is poor. On average, on the instances reported in Table 1, the value of the lower bound divided by the value of the optimal solution was 64.06%. In other words, the average error of the spanning tree lower bound was 35.94%. Further, as the average number of nodes in a cluster increases, the quality of the spanning tree lower bound deteriorates.

Kruskal's straightforward adaptation to the GMST is also used by Feremans et al. (2004) to generate upper bounds. We now propose two improvements to the straightforward adaptation of Kruskal's. First, we propose fixing a node to be in the generalized spanning tree and running Kruskal's adaptation (this is somewhat akin to choosing a starting node for Prim's adaptation). This may be accomplished by simply deleting all edges out of other nodes in the cluster, which the selected node is in, prior to the sort operation performed in Kruskal's. We propose running Kruskal's algorithm $|V|$ times, once for each possible choice of a node to be fixed in the solution, and selecting the best solution obtained.

The second improvement deals with the circumstance where the upper-bound heuristic is unable to find a feasible solution due to the fact that the underlying graph is not complete. Ideally, we would like to add an edge in Kruskal's adaptation only if the addition of the edge does not cause the heuristic to fail (i.e., make the problem infeasible). To do this, we need to answer the following question. Given a graph, clusters, and edges, does it contain a feasible GST? (When we add an edge, we have selected a node in a cluster. Thus, we can delete all other nodes in the cluster and edges emanating from them, and ask the question: Does the modified graph contain a feasible GST?) Unfortunately, in general, from the transformation given in Myung et al. (1995), even the recognition of whether a graph contains a generalized spanning tree is NP-complete. Consequently, we consider the following heuristic to handle infeasibility. If the algorithm results in an infeasible solution, we propose that the algorithm remove the most expensive edge added to the tree from the problem and run the heuristic again. We repeat this procedure until either a feasible GST is obtained or one cluster has no edges out of it. We call this improved version the *improved Kruskal's heuristic* (IKH).

If feasibility is not an issue (e.g., if the underlying graph is complete), then IKH runs in $\mathcal{O}(|V||E| + |V|^2 \log |V|)$ time. When feasibility is an issue, a crude running-time bound for a single iteration of IKH is $\mathcal{O}(|E|^2 + |E||V| \log |V|)$, and thus IKH runs in $\mathcal{O}(|E|^2|V| + |E||V|^2 \log |V|)$ time. However, this bound

Table 1 Comparison of Straightforward Adaptations of Kruskal's, Prim's, and Sollin's Upper Bound and the Spanning Tree Lower Bound

Clustering type	Number instances	Spanning tree lower bound avg error (%)	Upper bound procedures		
			Kruskal's avg error (%)	Prim's avg error (%)	Sollin's avg error (%)
center	37	37.13	8.27	13.10	7.99
grid, $\mu = 3$	28	22.22	5.05	7.94	5.22
grid, $\mu = 5$	28	32.95	10.65	12.41	10.70
grid, $\mu = 7$	28	42.34	15.87	17.18	15.64
grid, $\mu = 10$	29	44.39	15.90	18.56	15.67
Overall	150	35.94	11.00	13.83	10.89

Note. Comparison on 150 TSPLIB instances reported in §5 where the optimal solution is known. Kruskal's was best in 85 instances, Prim's was best in 41 instances, and Sollin's was best in 79 instances. There were 55 ties between Kruskal's and Sollin's.

is somewhat misleading, because when the graph is dense (i.e., $|E|$ is large), infeasibility is very unlikely. Furthermore, observe that in Kruskal's algorithm the most expensive edge in the tree constructed is the last edge that was added to it. Thus, instead of building a tree from scratch, we may simply delete the last edge added and continue to build the tree from there.

We also modified Prim's straightforward adaptation to the GMST to take care of the feasibility issue. In this adaptation, which we call the *improved Prim's heuristic* (IPH), we propose running Prim's algorithm $|V|$ times, once for each possible choice of starting node, and selecting the best solution obtained. To deal with the situation where the heuristic is unable to find a feasible solution, we delete the most recently added edge to the current tree and run the heuristic again.

If feasibility is not an issue (e.g., if the underlying graph is complete), then IPH runs in $\mathcal{O}(|V||E| + |V|^2 \log |V|)$ time. When feasibility is an issue, arguing as for IKH, a crude running-time bound for IPH is $\mathcal{O}(|E|^2|V| + |E||V|^2 \log |V|)$. Again this is somewhat misleading for the very same reason as in the case of IKH. Also, as in the case for IKH, we can delete the most recently added edge and continue to rebuild a tree from there.

In a similar fashion to IKH and IPH, we improved Sollin's adaptation for the GMST by fixing one node to be in the GST, running the adaptation $|V|$ times (once for each possible choice of a node to be fixed in the solution), and selecting the best solution obtained. To deal with the situation where the heuristic is unable to find a feasible solution, we delete the most recently added edge to the current solution and run the heuristic again. We call this adaptation the *improved Sollin's heuristic* (ISH). If feasibility is not an issue, then ISH runs in $\mathcal{O}(|V||E|K)$ time. When feasibility is an issue, arguing as for IKH, a crude running-time bound for ISH is $\mathcal{O}(|E|^2|V|K)$.

We compared the three upper bound heuristics, IKH, IPH, and ISH, on the 150 TSPLIB instances where the optimal solution is known. Table 2 presents these results. In contrast to the earlier results reported in Table 1, surprisingly, the results between Kruskal's, Prim's, and Sollin's are comparable. Compared to the results in Table 1, the average error improves markedly. Notice that when the average number of nodes belonging to a cluster is small, IKH and ISH are better than IPH. As the average number of nodes in a cluster increases, IPH does better than IKH and ISH. We note that, for the 150 instances in Table 2, the average running times for our implementations of IKH, IPH, and ISH were 0.27, 0.02, and 17.16 seconds, respectively.

Table 2 Comparison of Improved Adaptations of Kruskal's (IKH), Prim's (IPH), and Sollin's (ISH)

Clustering type	Number instances	IKH Average error (%)	IPH Average error (%)	ISH Average error (%)
center	37	4.80	6.81	4.82
grid, $\mu = 3$	28	3.42	5.06	3.60
grid, $\mu = 5$	28	6.48	6.45	6.59
grid, $\mu = 7$	28	8.98	7.43	8.67
grid, $\mu = 10$	29	8.43	6.76	8.07
Overall	150	6.34	6.52	6.27

Note. Comparison on 150 TSPLIB instances reported in §5 where optimal solution is known. IKH was best in 74 instances, IPH in 58, and ISH in 65. There were two ties between IKH, IPH, and ISH; and 45 ties between IKH and ISH.

3. Local-Search Procedure

Local search has been successfully used to find near-optimal solutions for a wide variety of combinatorial optimization problems (see Aarts and Lenstra 1997 for an up to date discussion of local search in combinatorial optimization). In this section, we propose a simple local-search heuristic for the GMST problem.

The local-search procedure starts with a randomly generated feasible GST (obtained by randomly selecting a node in each cluster and constructing an MST on the selected nodes) and then visits clusters in a wraparound fashion following a randomly defined order. It then replaces the node from the cluster being visited with another node from the same cluster and evaluates the cost of the new MST (any one of the three minimum spanning tree algorithms may be used for this purpose). It repeats this procedure, trying every other node in the same cluster, and selects the lowest-cost solution as the current solution. The local-search procedure stops when no further improvement is possible. This occurs when all clusters are visited in sequence with no improvement in the solution. We apply this local-search procedure using a prespecified number of starting solutions (referred to as X). The steps of the procedure are outlined below.

Local-Search Heuristic (LS):

Step 0. Specify the number of feasible solutions to be generated, X. Repeat Steps 1 through 3 X times.

Step 1. Randomly select a single node from each cluster. If the subgraph defined by the edges between the selected nodes is connected, apply any one of the minimum spanning tree algorithms to the subgraph defined on the selected nodes to build an MST. Otherwise, repeat Step 1.

Step 2. Randomly define an order in which clusters will be searched.

Step 3. Follow the order defined in Step 2 in visiting clusters. Repeat the following steps until K sequential cluster visits result in no improvement.

(a) While visiting a cluster, consider each node in the cluster as a replacement for the current node in

the cluster contained in the GST. Compute the cost of the solution for each replacement.

(b) Among the solutions computed in the previous step, identify the one giving the greatest improvement in the objective function (i.e., greatest reduction in solution cost). If there is an improvement, implement it by replacing the node representing the cluster, and update the current tree.

We briefly comment on the complexity of some of the steps of LS. Observe that the subgraph defined on the selected nodes contains K nodes and can be identified in $\mathcal{O}(|V|+|E|)$ time. With this observation, it is easy to show that Step 1 may be accomplished in $\mathcal{O}(|E|+K^2+K\log K)$ time if Kruskal's and Prim's are used to construct the spanning tree, and in $\mathcal{O}(|E|+K^2\log K)$ time if Sollin's is used to construct the spanning tree. A key part of Step 3 is to compute the cost of a minimum spanning tree when one node from the existing MST is deleted and a new node is added to the graph. Observe that when a node is deleted from an existing spanning tree the resulting forest may contain no edges (it is possible that all the edges in the MST are connected to the node that was deleted). Thus, to find the new MST we need to run an MST algorithm again, and the complexity is $\mathcal{O}(K^2+K\log K)$ (assuming Prim's or Kruskal's algorithm is used to generate the tree). If there are $|V_i|$ nodes in the cluster being visited, then Step 3 takes $\mathcal{O}(|V_i|(K^2+K\log K))$ time.

The running time of Step 3 can be improved based on the following claim. Given a graph $G=(V,E)$, let T_G denote the minimum spanning tree on G. Suppose we add a new node p to the graph, and edges $E_p=\{\{p,i\}: i\in V\}$. Let $G^p=(V\cup p, E\cup E_p)$ denote this new graph obtained by the addition of the new node p, and let T_{G^p} denote the minimum spanning tree on G^p.

Claim 1. *T_{G^p} does not contain any edges in $E\backslash T_G$.*

Proof. Without loss of generality, assume all edge costs are distinct. Suppose T_{G^p} contains an edge $\{i,j\}$ from $E\backslash T_G$. Path-optimality conditions for the minimum spanning tree assure us that the cost of every edge on the path, P_{ij}, from node i to node j in T_G is less than the cost of edge $\{i,j\}$. On the other hand, the cut-optimality conditions for the minimum spanning tree state that the cost of edge $\{i,j\}$ is less than the cost of every edge in the cut formed by deleting edge $\{i,j\}$ from T_{G^p}. Now, one of the edges in P_{ij} must be in this cut; thus, we have a contradiction. □

The above result may now be used to improve the running time of Step 3 as follows. First, construct the minimum spanning tree on the nodes other than the cluster being visited. Because there are $K-1$ nodes, the running time required for this is $\mathcal{O}(K^2+K\log K)$. Now, determine the minimum spanning tree when we replace the node in the current cluster. Based on the above result, we may compute the MST on a graph that contains the $K-1$ edges in the MST just computed, and the edges out of the node now being visited in the current cluster (there are at most $K-1$ edges, one to each node representing the other $K-1$ clusters). In other words, we need to compute the MST on a graph containing K nodes, and at most $2K-3$ edges. Hence finding this MST takes $\mathcal{O}(K+K\log K)$ time. Since we perform this calculation $|V_i|-1$ times while visiting cluster i, once for each potential replacement node, the overall running time is $\mathcal{O}(K^2+K\log K+|V_i|(K+K\log K))$, or $\mathcal{O}(K^2+|V_i|K\log K)$. We do not comment on the overall worst-case complexity of LS, as, in general, for any neighborhood structure the worst-case complexity analysis for local-search heuristics is bad, while their average-case complexity is good (see Tovey 1997).

Feremans (2001) also describes a local-search heuristic for the GMST problem. She uses her heuristic to improve upon feasible solutions found in her branch-and-cut procedure. Her local-search algorithm works as follows.

Feremans Local-Search Heuristic (FLS):
Visit the clusters once in increasing order of cluster number ($k=1,\ldots,K$) and perform the following steps during a visit to a cluster.

Step 0. Consider the set of nodes in the current solution. For this set, delete the node representing the cluster in the current solution, and add all the other nodes in the cluster to it. Create the graph consisting of this set of nodes and all edges between them.

Step 1. Apply the straightforward adaptation of Kruskal's to the graph defined in Step 0.

Step 2. If a lower-cost solution is found in Step 1, replace the current solution by this lower-cost solution.

Notice that LS explicitly tries every node in a cluster as a substitute for the current node representing the cluster in the solution. On the other hand, FLS considers all the other nodes in the cluster and applies the straightforward adaptation of Kruskal's. However, when Kruskal's is applied, the lowest-cost edge from among the nodes in the cluster will be selected first. Thus, FLS will select the node with the smallest-cost edge out of the cluster to represent the cluster being visited. Thus, it effectively tries only *one other node in the cluster as a substitute* for the current node representing the cluster. Consequently, we expect LS to provide better solutions than FLS. On the other hand, FLS visits the clusters in order once, and performs Kruskal's adaptation at most K times in Step 1. Consequently, we expect the running time of FLS to be significantly faster than LS. These expectations are confirmed in our computational experiments.

```
Begin
    t ⟵ 0
    initialize P(t)
    while (not termination-condition) do
        t ⟵ t + 1
        generate αP(t − 1) offspring using crossover operator on parents in P(t − 1)
        generate βP(t − 1) offspring using tree separator operator on parents in P(t − 1)
        generate γP(t − 1) offspring using random mutation operator on parents in P(t − 1)
        generate δP(t − 1) offspring using local search (mutation) operator on parents in P(t − 1)
        Select/Determine P(t) from generated offspring and P(t − 1)
    end
end
```

Figure 2 Steps of our Genetic Algorithm

4. Genetic Algorithm

Genetic algorithms are powerful local-search algorithms based on ideas of natural selection and genetics in nature. Genetic algorithms were originally proposed by Holland (1962), and have been successfully applied to numerous, hard combinatorial optimization problems. In this section, we describe a genetic algorithm that obtains high-quality solutions for the GMST problem.

An outline of our genetic algorithm is shown in Figure 2. We first generate an initial population of feasible solutions. In every generation in the genetic algorithm, we generate new solutions (offspring) using one of four operators—crossover, tree separation, (random) mutation, and local search (mutation). From among the parents and offspring produced in a generation, we select a subset of the solutions to survive to the next generation. This is continued until the termination condition is met, and the best solution obtained so far is output by the genetic algorithm.

We now elaborate on the details of our genetic algorithm.

Representation. We represent a generalized spanning tree by a chromosome that is an array of size K. In this representation, the ith entry (gene value) indicates the node selected to represent cluster i in the generalized spanning tree. Figure 3 shows the chromosome representation of the generalized spanning tree in Figure 1. We note that there are other possible representations, such as a binary string of size $|V|$ indicating nodes in the solution, or a binary string of size $|E|$ indicating the edges in the solution. The representation/schema provided here is compact, and is easy to construct and apply genetic operators to. (It is important to build compact representations, as Goldberg 1989 formally argues that short, low-order schema result in high-quality solutions.)

Initial Population. To generate an initial population, we select one node from each cluster randomly.

4	6	10	12	16

Figure 3 Chromosome Representation for the GST from Figure 1

We then construct a minimum spanning tree on these nodes. (Any of the three algorithms for the MST may be used. In our implementation, we use Prim's.) If there is no feasible spanning tree on the selected nodes, the chromosome is discarded. Otherwise, we record the fitness of the chromosome as the cost of the minimum spanning tree and add the chromosome to the initial population. We repeat this procedure until a prespecified initial population size is reached. Notice that the procedure generates an initial population that is feasible, and that this initial population may contain identical chromosomes.

Crossover. Our crossover operation, a standard one-point crossover, takes two parents and combines them to form two new solutions as follows. A crossing site is selected at random—possible crossing-site locations are immediately after the first gene, immediately after the second gene, and so on, until immediately after the $(K-1)$st gene. Two children are then obtained by taking the contents of the first parent before the crossing site together with the contents of the second parent after the crossing site, and the contents of the second parent before the crossing site together with the contents of the first parent after the crossing site. If there is no feasible tree over the nodes in a child chromosome, the child is discarded. Figure 4 illustrates the crossover operation.

Tree Separation. Tree separation creates two offspring from a single parent. It may be viewed as a self-reproduction genetic operator. The idea behind

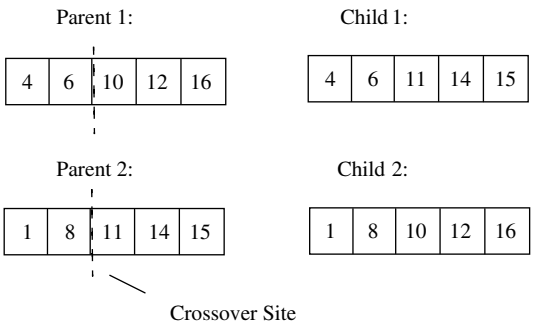

Figure 4 An Example of One-Point Crossover Operator

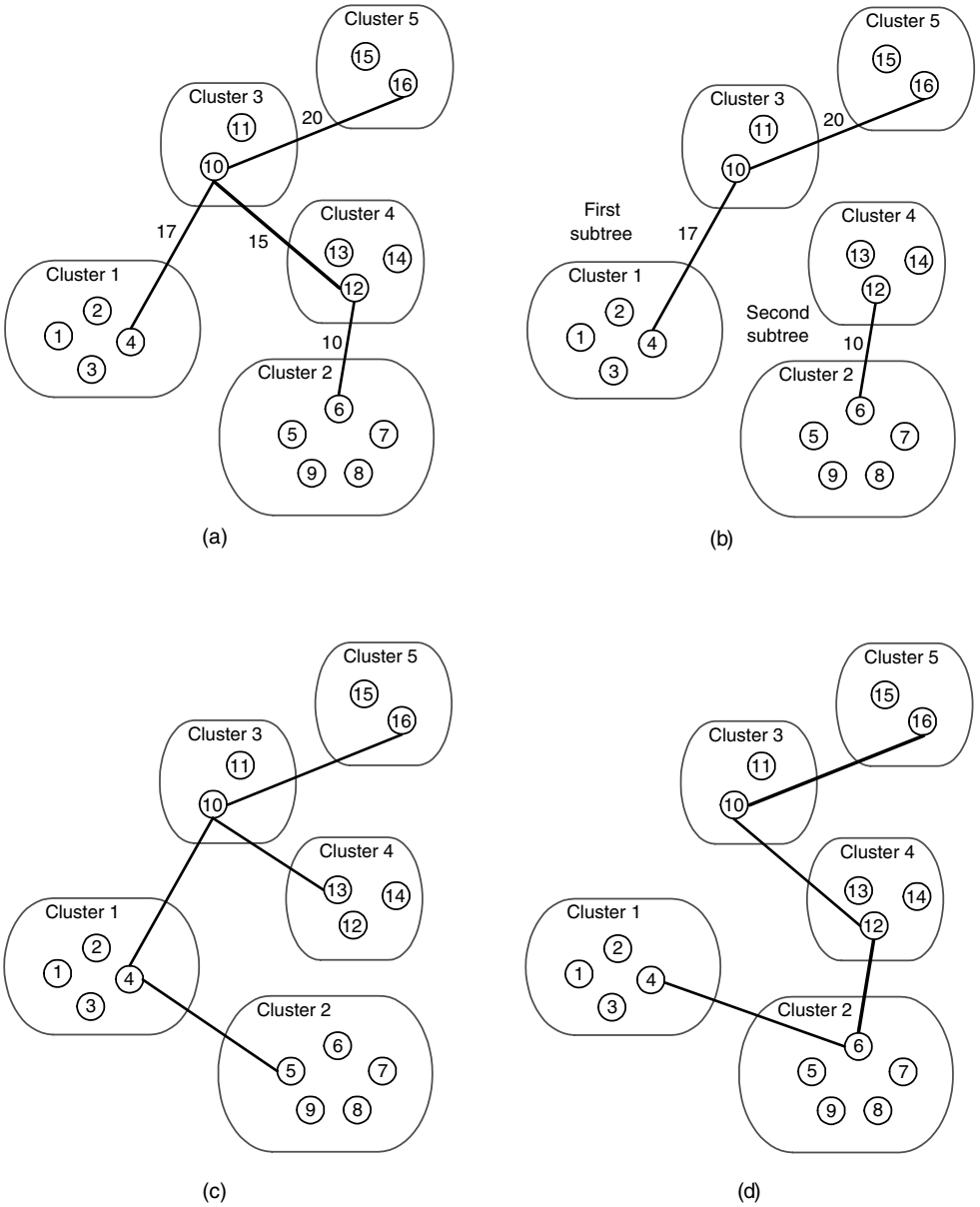

Figure 5 Example of the Tree Separation Operator

Note. In this example, we require each subtree to contain at least two nodes. (a) Initial tree. (b) Subtrees obtained by deleting the most expensive edge (10, 16), and second most expensive edge (10, 4), result in single-node subtrees, but subtrees obtained by removal of the third most expensive edge (10, 12) results in two trees containing at least two nodes. (c) First child created from parent in (b). (d) Second child created from parent in (b).

the tree-separation operator is to create two new children, each preserving some of the tree structure from the parent chromosome. To accomplish this, our genetic operator removes an edge in the parent tree to create two subtrees. Two children are then created by using each of these subtrees as a partial solution. Each one is completed using a straightforward adaptation of Prim's to obtain a generalized spanning tree. To ensure that each child has sufficient genetic material from a parent and contains the lower-cost edges from the parent, we start by removing the most expensive edge in the parent tree. If the number of nodes in each subtree is greater than or equal to a prespecified minimum number of nodes, two new chromosomes are created using the tree-completion procedure. Otherwise, we delete the second most expensive edge, and so on, until we obtain two subtrees, each containing more nodes than does the prespecified minimum. Figure 5 illustrates the tree-separation procedure with the minimum requirement of two nodes per subtree.

Random Mutation. Our random mutation operator randomly selects a gene in the chromosome and replaces it by another node in the same cluster. The replacement node is chosen from among the other nodes in the cluster at random (i.e., with equal probability).

Local Search. We apply local search on a chromosome as a type of mutation operation. In the local-search operator, a random ordering of clusters (or genes) is selected and the clusters are visited repeatedly (in the ordering selected initially) until no further improvement is possible (specifically, until all clusters are visited once sequentially without any improvement). When visiting a cluster, the local-search operator considers every other node in the cluster as a replacement for the current node representing the cluster, and it selects the node that provides the greatest improvement in the fitness (cost of the tree) of the chromosome.

Selection/Replacement. The selection procedure is an important step in the genetic algorithm. If the selective pressure is too strong (i.e., only the fittest individuals are selected to survive), then it is possible that some very fit chromosomes will dominate early in the search process and lead the algorithm to a local optimum, thus terminating the search. On the other hand, if the selective pressure is too weak (i.e., even individuals with low fitness survive), then the algorithm will traverse the solution space aimlessly.

Our selection procedure is as follows. At the end of each generation, from the offspring and parents we select a fraction θ of the population to survive and constitute the next generation based on the following criteria. We employ elitism to select the top 10% of $P(t)$. Recall, the chromosome-fitness function is the length of the minimum spanning tree defined over nodes in the chromosome. The remaining 90% of the population to be carried to the next generation is selected based on the ranking-probability distribution function (see Michalewicz 1996). (For example, if 50 individuals will survive to generation t, 5 are selected by elitism and 45 are selected using the ranking-probability distribution function.) Here the probability with which we select a chromosome with rank r is given by

$$\text{Prob}\{r\} = q(1-q)^{r-1} \times \frac{1}{1-(1-q)^{(|P(t-1)|+(\alpha+\beta+\gamma+\delta)|P(t-1)|)}},$$

where q represents a user-defined parameter that controls the selective pressure of the algorithm. This parameter takes values between 0 and 1, where higher values indicate more selective pressure. Note that $(\alpha+\beta+\gamma+\delta)|P(t-1)|$ represents the number of offspring generated using the genetic operators in generation $t-1$, and thus the exponent of $(1-q)$ in the denominator represents the size of the total population before the selection step.

5. Computational Experiments

We now report on our computational experiments with the LS heuristic and the genetic algorithm (GA) procedure. We coded both LS and GA in Microsoft Visual C++ on a Pentium III 800 MHz PC with 256 MB of RAM. We found that, on small instances, both LS and GA always found the optimal solution. Consequently, we tested our heuristics on two classes of *large* instances—those with edge costs that satisfy the triangle inequality, and those with random edge costs.

The instances where the edge costs satisfy the triangle inequality are from TSPLIB, and are identical to those in Feremans (2001) and Feremans et al. (2004). These are actually a subset of the instances described in Fischetti et al. (1997). They contain all problems from TSPLIB 2.1 (see Reinelt 1991) with 48 to 226 nodes (i.e., $48 \le |V| \le 226$) and two 47 node problems, spain47 and europe47, generated by Fischetti et al. (1997). (The naming convention of the TSPLIB problems identifies the number of nodes in the problem at the end of the problem name.) Two clustering procedures were used by Fischetti et al. (1997) to generate the clusters. For geographical instances, spain47 (cities in Spain), europe47 (European cities), gr96 (African cities), gr137 (American cities), and gr202 (European cities), clusters are chosen in a natural way where clusters correspond to a country (or region for spain47). For the remaining problems, Fischetti et al. (1997) develop two clustering procedures to simulate geographical regions. The first procedure works roughly as follows. It fixes the number of clusters K to be $\lceil |V|/5 \rceil$ and then determines K centers by considering K nodes that are as far as possible from one another. The clusters are then obtained by assigning nodes to their closest cluster. The second procedure, called *grid clusterization*, works roughly as follows. It constructs an $H \times H$ square grid, and nodes contained within a box of this square grid are assigned to a cluster. The size of the square grid H is selected to be the smallest H, so that the number of nonempty boxes in this $H \times H$ square grid is $\ge |V|/\mu$ (where μ is a user-defined parameter that may be thought of, roughly, as the average number of nodes in a cluster).

The large random instances were generated using the following procedure, which is similar to the problem-generation technique described in Dror et al. (2000). We are given the number of nodes $|V|$, the number of edges $|E|$, and the number of clusters K, as inputs. We first partition nodes into clusters as follows. For each cluster, we generate a random number in the range $[0, 1]$. The ratio of this random number to the sum of random numbers over all clusters defines the percentage of nodes that will be assigned to a given cluster. We then add edges to the graph. First, we arbitrarily construct a feasible generalized spanning tree and create edges corresponding to these edges. We then add $|E| - |V| + 1$ edges (that are between clusters) to the graph. It is ensured

that each node has at least one edge incident to it. Edge costs are integer and are generated uniformly in the range 1 to 50. Observe that edge costs can be randomly assigned since edge cost in the random instances does not represent distance between two nodes. Instead, it may represent a cost that depends on compatibility of two nodes to be connected.

Parameter Settings for LS and GA. After initial testing, we selected the GA parameters as follows. The size of the initial population $P(0)$ was set to 500, while α, β, γ, and δ were set to 0.1, 0.1, 0.7, and 0.1, respectively. Thus, in each generation, 500 offspring were generated. In the tree-separation operator, we set 20% of the nodes as the prespecified required number of nodes. Additionally, we considered dropping only the three most expensive edges. If that does not result in a feasible solution, a new parent node is selected for the tree-separation operation. The fraction θ of the population to survive was set to 0.5, and the user-defined parameter q in the rank-based probability distribution function was set to 0.15. The stopping criterion was 25 generations.

To compare LS fairly with GA, we provided LS the same set of starting solutions as GA. Thus, we set the number of starting solutions $X = 500$.

TSPLIB Instances. We now report on our computational experiences with the TSPLIB instances. For these problems, we obtained the optimal solutions and best lower and upper bounds (when no optimal solution was available) from Feremans (2001). There are a total of 169 instances, for which optimal solutions are known in 150 instances. The results are presented in Tables 3 through 6.

The results of LS and GA on the TSPLIB instances are quite compelling. Of the 150 instances for which the optimal solution is known (see Tables 3 through 5), GA found the optimal solution in 149 instances, while LS found the optimal solution in 145 out of 150 instances. In all the 19 instances where the optimal solution is not known (see Table 6), both LS and GA improved upon the best previously known solutions. In these instances, the percentage improvement ranged from 0.01% to 5.31%, with an average improvement of 2.43% for LS and 2.51% for GA. In 6 of these 19 instances, GA found a better solution than LS. It is interesting to note that, over all 169 instances, LS never found a better solution than GA, while GA improved upon LS in 10 instances. On the other hand, the running time of LS is faster than GA. On average, GA took about twice as long as LS, albeit with very impressive results.

For the 19 instances where the optimal solution is unknown, the average gap of the GA solution compared to the lower bound is 12.35%. On the other hand, on the 150 instances where the optimal solution is known, the GA is 0.0001% above optimality. Given the experience on the 150 instances where the optimal solution is known, we believe that lower bounds for the unsolved instances are quite far from the optimal integer solution. We strongly believe that the optimal solution for these instances is equal to, or very close to, the GA solution.

Table 3 Computational Results for LS and GA on TSPLIB Instances with the Center Clustering Procedure Where the Optimal Solution Is Known

Problem name	K	$\|E\|$	Optimal solution	LS time (sec)	GA time (sec)
spain47	15	985	2,393	2	7
europ47	27	1,042	13,085	4	13
gr96	50	4,463	306	27	57
gr137	35	8,251	209	19	39
gr202	34	19,018	135	28	59
att48	10	1,010	10,923	1	6
gr48	10	1,017	1,282	1	6
hk48	10	995	4,119	1	6
eil51	11	1,158	132	1	6
brazil58	12	1,464	9,206	2	7
st70	14	2,248	233	3	8
eil76	16	2,660	186	3	9
pr76	16	2,661	46,514	4	11
gr96	20	4,292	221	6	14
rat99	20	4,609	402	6	15
kroa100	20	4,727	7,982	7	15
krob100	20	4,716	8,111	6	15
kroc100	20	4,714	8,041	7	15
krod100	20	4,716	7,643	7	15
kroe100	20	4,702	8,164	7	15
rd100	20	4,703	2,779	7	15
eil101	21	4,776	204	7	16
lin105	21	5,130	6,728	7	17
pr107	22	5,441	20,398	8	18
gr120	24	6,820	2,255	11	22
pr124	25	7,315	30,174	11	23
bier127	26	7,191	58,150	14	25
pr136	28	8,879	34,104	18	31
gr137	28	8,892	329	15	32
pr144	29	9,952	40,055	18	34
kroa150	30	10,809	9,815	23	39
krob150	30	10,807	10,048	22	39
pr152	31	11,080	39,109	23	41
u159	32	12,031	18,723	26	45
rat195	39	18,478	751	47*	79
kroa200	40	19,409	11,634	55	86
krob200	40	19,430	11,244	52	84

Note. Both procedures found the optimal solutions except as noted.
*LS found a nonoptimal solution with value 753.

Large Random Instances. We now report on our computational experience with large random instances. For these instances, to determine the quality of the solution produced by LS and GA, we solved the linear-programming relaxation of the mixed-integer programming formulation proposed by Raghavan (2002) and described in the appendix.

When we solved the linear-programming relaxation, we actually solved the dual of the linear-programming relaxation. The reason for this is twofold.

Table 4 **Computational Results for LS and GA on TSPLIB Instances with Grid Clustering Procedure, $\mu = 3$ and 5, Where Optimal Solution Is Known**

	$\mu = 3$					$\mu = 5$				
Problem name	K	$\|E\|$	Optimal solution	LS time (sec)	GA time (sec)	K	$\|E\|$	Optimal solution	LS time (sec)	GA time (sec)
att48	18	1,062	16,521	3	9	13	1,025	13,189	2	6
eil51	25	1,235	242	4	12	16	1,214	158	2	8
st70	24	2,329	297	6	14	16	2,277	214	3	9
eil76	36	2,789	306	13	28	16	2,695	149	3	9
pr76	31	2,770	58,038	11	20	16	2,661	29,788	3	9
gr96	33	4,413	298	13	28	22	4,315	234	7	16
rat99	36	4,753	521	18	34	25	4,692	410	9	19
kroa100	43	4,844	11,914	23	46	23	4,748	8,054	9	17
krob100	44	4,854	12,561	28	49	25	4,743	7,880	11	19
kroc100	42	4,847	12,284	24	44	25	4,762	8,084	11	20
krod100	42	4,846	11,827	25	45	24	4,724	8,741	9	18
kroe100	42	4,846	12,292	25	44	25	4,738	8,401	9	19
rd100	36	4,801	3,978	18	34	24	4,742	3,077	8	18
eil101	36	4,912	295	18	35	25	4,812	217	10	19
lin105	42	5,340	9,280	31	54	30	5,237	7,410	14	27
pr107	45	5,542	23,290	22	44	22	5,438	19,877	8	16
pr124	42	7,440	37,837	31	57	25	7,219	27,156	12	22
bier127	50	7,729	71,221	42	75	26	7,173	58,989	14	23
pr136	60	9,064	52,817	61[a]	107	34	8,888	37,735	25	40
gr137	49	9,122	391	43	80	32	8,908	338	18	35
pr144	48	10,084	43,725	39	74	30	9,928	36,279	18	33
kroa150	57	11,005	14,050	81	119	36	10,868	10,101	35	50
krob150	56	10,982	13,845	66	109	36	10,870	9,780	31	50
pr152	54	11,254	44,253	56	99	33	11,083	38,143	25	42
u159	58	12,321	24,214	73	115	32	12,046	17,059	25	40
rat195	51	18,745	1,111	190	290	49	18,600	796	77	114
kroa200	72	19,636	14,881	155[b]	259	47	19,464	11,628	78	108
krob200	76	19,661	15,320	186	336	48	19,511	11,113	78[c]	114[d]

Note. Both procedures found the optimal solutions, except as noted.

[a]LS found a nonoptimal solution with value 52,824.

[b]LS found a nonoptimal solution with value 14,897.

[c]LS found a nonoptimal solution with value 11,115.

[d]GA found a nonoptimal solution with value 11,115.

First, in most instances, solving the dual is faster than solving the primal. Second, *any* feasible solution to the dual is a valid lower bound for the problem. Thus, even if we terminated (due to excessively long running times) Phase II of the simplex method, we have a valid bound for the GMST. Further, since the edge costs in the problem are integer, we may round up the value of the lower bound (as the optimum integer feasible solution will have an integer objective value) to obtain an improved lower bound for the GMST.

In Table 7, we report on our computational experience with large random instances. We solved the dual of the LP relaxation of the MIP formulations using CPLEX 7.1 on a Sun Microsystems Enterprise 250 with 2×400 MHz processors and 2 GB of RAM. The running times (in seconds) to obtain these optimal solutions were quite large. Several instances required weeks of CPU time. Further, in many instances, we manually terminated Phase II of the simplex when the objective was within one unit of the solution found by GA or if the running time was inordinately long. Again, the results of LS and GA are quite compelling. Out of the 42 instances, GA was strictly better than LS in 5 instances, while LS was strictly better than GA in 1 instance.

The remaining 36 instances were ties. Of the 42 instances, we are able to show optimality of the LS or GA solution in 37 instances, with LS finding the optimal solution in 34 instances and GA finding the optimal solution in 36 instances. In the remaining five instances (where we cannot show that LS or GA obtained the optimal solution), the objective value of GA is within one unit of the lower bound in four instances and within two units of the lower bound in one instance. It is quite possible that LS or GA obtained optimal solutions on these instances. For example, in the instance with $K = 20$, $|V| = 160$, and $|E| = 5{,}000$, we solved the mixed-integer programming formulation to optimality, and found that the optimal solution was 35, indicating that LS and GA both found the optimal solution. We did not attempt to do this on the remaining four instances (where the

Table 5 Computational Results for LS and GA on TSPLIB Instances with Grid Clustering Procedure, $\mu = 7$ and 10, Where Optimal Solution Is Known

	$\mu = 7$					$\mu = 10$				
Problem name	K	$\|E\|$	Optimal solution	LS time (sec)	GA time (sec)	K	$\|E\|$	Optimal solution	LS time (sec)	GA time (sec)
att48	7	947	6,667	1	5	7	947	6,667	1	5
eil51	9	1,143	100	1	6	9	1,143	100	1	5
st70	16	2,277	214	3	9	9	2,160	147	1	6
eil76	16	2,695	149	3	9	9	2,545	94	1	6
pr76	16	2,661	29,788	3	9	9	2,532	20,501	1	6
gr96	15	4,170	186	3	9	15	4,170	186	3	9
rat99	16	4,583	308	4	11	16	4,583	308	4	11
kroa100	16	4,647	5,987	5	11	16	4,647	5,987	4	11
krob100	16	4,652	6,058	5	11	16	4,652	6,058	5	11
kroc100	16	4,651	5,534	5	11	16	4,651	5,534	5	11
krod100	16	4,647	5,904	5	11	16	4,647	5,904	5	11
kroe100	16	4,607	6,450	5	11	16	4,607	6,450	5	11
rd100	16	4,606	2,287	4	11	16	4,606	2,287	4	11
eil101	16	4,757	141	4	11	16	4,757	141	4	11
lin105	16	5,005	4,542	5	11	16	5,005	4,542	5	11
pr107	16	5,322	17,547	5	11	12	5,196	16,754	3	8
pr124	19	7,077	23,164	7	15	14	6,896	18,554	4	10
bier127	19	7,039	52,097	7	15	14	6,240	43,778	4	10
pr136	20	8,724	22,541	8	17	16	8,548	21,732	5	13
gr137	22	8,746	264	9	20	15	8,504	197	4	12
pr144	21	9,762	33,947	9	19	16	9,508	32,510	5	13
kroa150	25	10,703	7,944	15	27	16	10,506	5,229	7	14
krob150	25	10,725	7,293	17	28	16	10,455	5,494	7	14
pr152	24	11,008	35,429	14	25	16	10,828	33,340	6	14
u159	23	11,896	12,659	14	23	23	11,896	12,659	14	23
rat195	36	18,454	639	44[a]	66	25	18,225	482	20	34
kroa200	35	19,335	9,640	45	63	25	19,100	6,895	23	35
krob200	36	19,362	9,742	45	67	25	19,082	6,922	24	36
pr226	—	—	—	—	—	27	24,137	43,389	22	40

Note. Both procedures found the optimal solutions, except as noted.
[a]LS found a nonoptimal solution with value 648.

optimal integer solution is not known) as it was computationally prohibitive to do so.

Comparing LS Against FLS. We finally report on some computations we performed to compare LS against FLS. As we indicated in §3, FLS considers only one other node as a substitute for the current node in the cluster, while LS considers all other nodes in the cluster as possible substitutes. We wanted to see to what extent this difference in neighborhood structure affected the quality of the solutions (and the ability of the local-search heuristics to find the optimal solution). We coded FLS (in Microsoft Visual C++ on the same machine as LS) and ran it on the 211 instances with the identical set of 500 starting solutions for each problem instance as LS. We found that out of the 211 instances, FLS found the optimal solution in only 24 instances, as compared to LS, which found the optimal solution in 179 instances. In 187 instances, the solution obtained by LS is strictly better than the solution obtained by FLS, while FLS never obtained a better solution than LS. Over the 187 instances for which the optimal solution is known, FLS generated solutions that are, on average, 3.28% from optimality (compared to 0.07% for LS). For the 211 instances, the average running times were 23.10 seconds for LS, and 2.09 seconds for FLS.

We note that, compared to LS, FLS performs an abbreviated search, visiting each cluster just once. Since we want to compare the difference in neighborhoods, we developed an improved version of FLS, called *complete FLS*, where local search is performed until no further improvement is possible. Complete FLS is identical to LS, except that the neighborhood structure is as in FLS. We coded complete FLS (in Microsoft Visual C++ on the same machine as LS) and also ran it on the 211 instances (with the identical set of 500 starting solutions for each problem instance as LS). On examining the results for complete FLS, we found it is better than FLS, but its results fall short of LS. Out of the 211 instances, complete FLS found the optimal solution in 83 instances. Further, in 121 instances, LS is strictly better than complete FLS, while complete FLS is strictly better than LS in 2 instances. Over the 187 instances for which the opti-

Table 6 Computational Results for LS and GA on TSPLIB Instances Where the Optimal Solution Is Unknown

Problem name	K	$\|E\|$	Lower bound	Previous best	LS		GA	
					Soln	Time (sec)	Soln	Time (sec)
TSPLIB instances, center clustering								
d198	40	18,841	5,743	7,232	7,044	51	7,044	84
gr202	41	19,532	217	250	244	43	243	85
ts225	45	24,650	62,131	62,506	62,400	69	62,315	115
pr226	46	24,626	50,206	55,971	55,515	59	55,515	115
TSPLIB instances, grid clustering, $\mu = 3$								
d198	67	19,101	7,129	8,599	8,285	131	8,283	216
gr202	68	19,826	265	302	293	105	293	220
ts225	75	24,900	78,028	81,962	79,085	182	79,019	298
pr226	84	25,118	44,078	63,148	62,527	174	62,527	321
TSPLIB instances, grid clustering, $\mu = 5$								
d198	40	18,772	5,775	7,291	7,098	51	7,098	79
gr202	41	19,303	202	243	234	40	232	78
ts225	45	24,726	56,018	62,242	60,713	70	60,659	107
pr226	50	24,711	55,800	56,822	56,721	67	56,721	118
TSPLIB instances, grid clustering, $\mu = 7$								
d198	32	18,372	5,273	6,759	6,501	32	6,501	52
gr202	31	18,872	180	207	203	27	203	49
ts225	35	24,544	50,281	53,661	50,813	47	50,813	69
pr226	33	24,355	47,965	48,254	48,249	34	48,249	59
TSPLIB instances, grid clustering, $\mu = 10$								
d198	25	18,149	4,999	6,351	6,185	20	6,185	34
gr202	21	17,904	161	185	177	12	177	26
ts225	25	24,300	40,298	41,162	40,339	22	40,339	37

mal solution is known, complete FLS generated solutions that were, on average, 0.55% from optimality. The average running time for complete FLS over the 211 instances was 6.87 seconds.

These results confirm our earlier assertion that LS is expected to provide better results than FLS, while the running time of FLS is expected to be significantly faster than LS. They also show that the neighborhood considered by LS is very effective in converging to the global optimum, while the neighborhood considered by FLS generally gets stuck at a locally optimal solution.

6. Conclusions

In this paper, we presented two heuristic search techniques—local search and a genetic algorithm—for the GMST problem. We also presented simple lower and upper bound heuristics for the GMST problem. There are several noteworthy contributions in this paper that we now summarize.

We considered the three well-known algorithms for the MST problem, and considered various approaches of adapting them to the GMST problem. In adapting these algorithms (as heuristics) to the GMST problem, we showed how to deal with infeasibility, when the graph is not complete, in a computationally effective manner. Feremans (2001) and Feremans et al. (2004) assert that Kruskal's adaptation to the GMST problem dominates Prim's and Sollin's adaptations. However, we showed that by repeating Kruskal's, Prim's, and Sollin's adaptation $|V|$ times, once each for a choice of a particular node in the generalized spanning tree, the quality of the solution produced by all three heuristics improves substantially and all three heuristics are competitive (i.e., neither dominates the other two). In our computational experience, when the number of nodes in a cluster is small, Kruskal's adaptation does better, while Prim's adaptation does better when the number of nodes in a cluster is large.

In developing our local-search procedure, we showed that LS searches the neighborhood of a generalized spanning tree more effectively than does FLS. Specifically, we showed that FLS considers only one other node in the cluster as a substitute for the current node representing the cluster, while LS considers every other node in the cluster as a potential substitute. We also showed that this neighborhood structure is particularly effective, compared to the neighborhood structure proposed by Feremans et al. (2004).

Finally, we tested our heuristics LS and GA on a test set of 211 large problem instances. Our results were quite compelling, and indicated that both LS and GA rapidly generate optimal or near-optimal solutions for the GMST problem.

Table 7 Computational Results for LS and GA on Large Random Instances

Problem characteristics			Dual of LP relax		Lower	LS		GA	
K	$\|V\|$	$\|E\|$	LP soln	Time (sec)	bound†	Soln	Time (sec)	Soln	Time (sec)
15	120	2,000	35	3,541	**35**	35	5	35	10
		3,000	23	978	**23**	23	5	23	10
		6,000	20.5*	7,477	**21**	21	5	21	10
	150	3,000	30	13,724	**30**	30	6	30	14
		5,000	24	12,858	**24**	24	6	24	12
		9,000	18.71	142,245	**19**	19	6	19	12
	180	4,000	25.83*	25,874	**26**	26	7	26	14
		7,000	16	4,304	**16**	16	7	16	14
		14,000	15	51,000	**15**	15	7	15	13
20	120	1,500	47	2,209	**47**	47	8	47	15
		3,000	31	4,077	**31**	31	8	31	15
		6,000	26.8	55,947	**27**	27	7	27	15
	160	3,000	36	24,428	**36**	38	11	36	19
		5,000	34	54,778	34	35	11	35	19
		10,000	23	120,879	**23**	23	10	23	18
	200	5,000	35.47*	63,539	**36**	36	14	36	23
		10,000	28.32*	366,275	**29**	29	14	29	23
		15,000	20.41*	84,765	**21**	21	12	21	22
25	150	3,000	54	10,428	**54**	54	16	54	26
		6,000	41	51,957	**41**	43	14	41	25
		9,000	33.09*	102,743	**34**	34	14	34	25
	200	5,000	41	61,085	**41**	41	22	41	33
		10,000	37.38	770,168	38	39	20	39	33
		15,000	26	142,046	**26**	26	19	26	32
30	120	2,000	73.59*	17,704	**74**	74	16	74	28
		3,000	55.2*	17,803	**56**	56	16	56	29
		6,000	42	62,476	**42**	42	14	42	28
	150	3,000	63	17,945	**63**	63	22	63	35
		6,000	47.5*	82,436	**48**	48	20	48	34
		9,000	39.33	421,907	40	41	19	41	34
	180	4,000	59	120,860	**59**	59	27	59	42
		7,000	47	193,727	**47**	47	26	48	40
		14,000	33.18*	192,303	**34**	34	23	34	40
40	120	1,500	117.33*	5,650	**118**	118	29	118	48
		3,000	73	9,405	**73**	73	29	73	47
		6,000	55	72,261	**55**	55	26	55	46
	160	3,000	91.19*	13,859	**92**	92	41	92	62
		5,000	78.75*	99,332	**79**	80	39	79	62
		10,000	53.17*	110,445	**54**	54	37	54	60
	200	5,000	86.88	532,824	87	91	51	89	78
		10,000	58.41*	1,111,914	59	61	48	60	78
		15,000	45.02*	1,702,291	**46**	46	44	46	74

*These problems were interrupted before the dual of the linear-programming relaxation was solved to optimality.
†Bold values identify instances where the lower bound is equal to the upper bound.

Acknowledgments

The authors thank Dr. Feremans for providing the TSPLIB data set. They are also grateful to an anonymous referee for suggestions concerning Sollin's adaptation.

Appendix

Modeling the GMST as a Steiner Tree with Degree Constraints

We first convert the undirected graph to a directed graph by replacing each edge $\{i, j\}$ by two directed arcs (i, j) and (j, i). The costs of arc (i, j) and (j, i), denoted by $c_{ij} = c_{ji}$, are equal to the cost of the undirected arc. Let the clusters be numbered $1, 2, \ldots, K$. We add nodes and connect them to the existing graph as follows.

1. Arbitrarily select a cluster r. Add a vertex s_r, and connect it to the selected cluster r by adding arcs (s_r, i) for all $i \in V_r$. The cost of these arcs is $c_{s_r i} = 0$.

2. For each cluster $k = 1, \ldots, K$, $k \neq r$, create a sink node t_k. Add arcs (i, t_k) for all $i \in V_k$, with cost $c_{it_k} = 0$. Let T denote the set of sink nodes created.

Consider the following variant of the directed Steiner tree problem on the transformed graph. Let s_r be the root node,

and the set of sink nodes T are the required nodes. We want a minimum-cost directed Steiner tree, with root node s_r, and containing all the nodes in T. Additionally, we need to ensure that exactly one node in each cluster is in the directed Steiner tree (as a Steiner node).

The following standard multicommodity flow model applies to the problem. Let y_{ij} be one if arc (i, j) is in the design, and be zero otherwise. Define a set of commodities H in the model as follows. For each $t_k \in T$, create a commodity $h \in H$ with origin s_r and destination t_k. Let $O(h)$ denote the origin of commodity h, $D(h)$ denote the destination of commodity h, and f_{ij}^h denote the flow of commodity h on arc (i, j). Given a node set S, we denote the set of arcs directed into this node set as $\delta^-(S)$, and the set of arcs directed out of this node set as $\delta^+(S)$.

Directed Flow-Formulation for the GMST Problem:

$$\text{Minimize} \quad \sum_{(i,j)\in A} c_{ij} y_{ij} \tag{1}$$

$$\text{subject to:} \quad \sum_{(j,i)\in\delta^-(\{i\})} f_{ji}^h - \sum_{(i,l)\in\delta^+(\{i\})} f_{il}^h = \begin{cases} -1 & \text{if } i = O(h); \\ 1 & \text{if } i = D(h); \\ 0 & \text{otherwise;} \end{cases}$$

$$\text{for all } i \in V \text{ and } h \in H \tag{2}$$

$$f_{ij}^h \le y_{ij} \quad \text{for all } (i, j) \in A \text{ and } h \in H \tag{3}$$

$$\sum_{(i,j)\in\delta^-(V_k)} y_{ij} = 1 \quad \text{for } k = 1, \ldots, K,\ k \ne r \tag{4}$$

$$y_{ij} + y_{ji} \le 1 \quad \text{for all } \{i, j\} \in E \tag{5}$$

$$f_{ij}^k, f_{ji}^k \ge 0 \quad \text{for all } \{i, j\} \in E \text{ and } h \in H \tag{6}$$

$$y_{ij} \in \{0, 1\} \quad \text{for all } (i, j) \in A. \tag{7}$$

In this formulation, constraints (4) ensure that the indegree of each cluster is 1, while the remaining constraints are as in the standard multicommodity flow formulation for the Steiner tree problem (Wong 1984). The above formulation is equivalent to the multicommodity flow formulation described by Myung et al. (1995) for the GMST. Myung et al. introduce node variables in their formulation to represent whether a node is selected to be in the GMST, while we expand the graph to create additional arc variables. In our computational experiments, we found that, although the two formulations are equivalent as linear programs, CPLEX solves the above formulation significantly faster.

References

Aarts, E. H. L., J. K. Lenstra. 1997. *Local Search in Combinatorial Optimization*. John Wiley and Sons, Chichester, UK.

Ahuja, R. K., T. L. Magnanti, J. B. Orlin. 1993. *Network Flows: Theory, Algortihms and Applications*. Prentice-Hall, Englewood Cliffs, NJ.

Dror, M., M. Haouari, J. Chaouachi. 2000. Generalized spanning trees. *Eur. J. Oper. Res.* **120** 583–592.

Feremans, C. 2001. *Generalized Spanning Trees and Extensions*. Ph.D. thesis, Institut de Statistique et de Recherche Opérationnelle, Université Libre de Bruxelles, Bruxelles, Belgium.

Feremans, C., M. Labbé, G. Laporte. 2002. A comparative analysis of several formulations for the generalized minimum spanning tree problem. *Networks* **39** 29–34.

Feremans, C., M. Labbé, G. Laporte. 2004. The generalized minimum spanning tree problem: Polyhedral analysis and branch-and-cut algorithm. *Networks* **43** 71–86.

Fischetti, M., J. J. Salazar-Gonzalez, P. Toth. 1997. Symmetric generalized traveling salesman problem. *Oper. Res.* **45** 378–394.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization Machine Learning*. Addison Wesley Longman, Inc., Reading, MA.

Holland, J. H. 1962. Outline for a logical theory of adaptive systems. *J. Association Comput. Machinery* **3** 297–314.

Michalewicz, Z. 1996. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Heidelberg, Germany.

Myung, Y. S., C. H. Lee, D. W. Tcha. 1995. On the generalized minimum spanning tree problem. *Networks* **26** 231–241.

Pop, P. C., W. Kern, G. J. Still. 2000. The generalized minimum spanning tree problem. Technical report, Department of Operations Research and Mathematical Programming, University of Twente, Twente, The Netherlands.

Pop, P. C., W. Kern, G. J. Still. 2001. An approximation algorithm for the generalized minimum spanning tree problem with bounded cluster size. Technical report, Department of Operations Research and Mathematical Programming, University of Twente, Twente, The Netherlands.

Raghavan, S. 2002. On modeling the generalized minimum spanning tree. Technical report, The Robert H. Smith School of Business, University of Maryland, College Park, MD.

Reinelt, G. 1991. TSPLIB—A traveling salesman problem library. *INFORMS J. Comput.* **3** 376–384.

Tovey, C. A. 1997. Local improvement on discrete structures. E. H. L. Aarts, J. K. Lenstra, eds. *Local Search in Combinatorial Optimization*. John Wiley and Sons, Chichester, UK, 57–90.

Wong, R. T. 1984. A dual ascent approach for Steiner tree problems on a directed graph. *Math. Programming* **28** 271–287.