Int. J. Appl. Math. Comput. Sci., 2004, Vol. 14, No. 3, 385–396

CODINGS AND OPERATORS IN TWO GENETIC ALGORITHMS FOR THE LEAF-CONSTRAINED MINIMUM SPANNING TREE PROBLEM

BRYANT A. JULSTROM*

* Department of Computer Science, St. Cloud State University
St. Cloud, MN, 56301, USA
e-mail: julstrom@stcloudstate.edu

The features of an evolutionary algorithm that most determine its performance are the coding by which its chromosomes represent candidate solutions to its target problem and the operators that act on that coding. Also, when a problem involves constraints, a coding that represents only valid solutions and operators that preserve that validity represent a smaller search space and result in a more effective search. Two genetic algorithms for the leaf-constrained minimum spanning tree problem illustrate these observations. Given a connected, weighted, undirected graph G with n vertices and a bound ℓ, this problem seeks a spanning tree on G with at least ℓ leaves and minimum weight among all such trees. A greedy heuristic for the problem begins with an unconstrained minimum spanning tree on G, then economically turns interior vertices into leaves until their number reaches ℓ. One genetic algorithm encodes candidate trees with Prüfer strings decoded via the Blob Code. The second GA uses strings of length $n-\ell$ that specify trees' interior vertices. Both GAs apply operators that generate only valid chromosomes. The latter represents and searches a much smaller space. In tests on 65 instances of the problem, both Euclidean and with weights chosen randomly, the Blob-Coded GA cannot compete with the greedy heuristic, but the subset-coded GA consistently identifies leaf-constrained spanning trees of lower weight than the greedy heuristic does, particularly on the random instances.

Keywords: evolutionary codings, leaf-constrained spanning trees, Prüfer strings, Blob Code, fixed-length subsets

1. Introduction

An evolutionary algorithm (EA) is a probabilistic search heuristic that replicates the defining features of biological evolution: reproduction with variation, selection based on fitness, and repetition. An EA maintains a population of data structures, called chromosomes, that encode candidate solutions to its target problem. Attached to each chromosome is its fitness, a numerical value that indicates the quality of the solution the chromosome represents. The algorithm selects chromosomes to survive or reproduce so that those with better fitness are more likely to be selected. Crossover, also called recombination, combines genetic information from two parent chromosomes. Mutation randomly modifies one parent chromosome. When the EA has generated enough offspring, they replace their parents and the process continues. As these generations succeed each other, chromosomes that represent better solutions evolve.

The several kinds of evolutionary algorithms are distinguished by the problems to which they are applied, the codings by which their chromosomes represent candidate solutions, the operators they apply to those chromosomes, and how they perform and use selection. Genetic algorithms (GAs) are most often applied to problems of combinatorial optimization, whose solutions they encode as strings of symbols. They apply crossover and mutation operators to generate offspring, and they select chromosomes from the population to reproduce via these operators.

The interaction between the coding by which an EA represents candidate solutions and the operators that generate offspring from the existing chromosomes is the single most important factor in determining whether the EA searches effectively. Also, when a problem involves constraints, a coding that represents only valid solutions and operators that preserve that validity can make the search space considerably smaller and the search correspondingly more effective.

Genetic algorithms for the leaf-constrained minimum spanning tree problem illustrate these observations. Given a connected, weighted, undirected graph G and a bound ℓ, this problem, which Section 2 describes in detail, seeks a spanning tree on G with at least ℓ leaves and minimum total weight among all such trees.

A recent greedy heuristic for the leaf-constrained minimum spanning tree problem begins with an unconstrained minimum spanning tree on G. One step of the heuristic turns an interior vertex into a leaf, forms a minimum spanning tree on the remaining interior vertices, and connects each leaf, including the new one, to the nearest interior vertex. The interior vertex chosen to become a leaf is the one for which the weight of this tree is smallest. This step is repeated until the number of leaves reaches ℓ. Section 3 describes this heuristic in detail and compares it with an earlier one for the problem.

Two evolutionary codings represent only spanning trees that satisfy the leaf constraint; that is, that have at least ℓ leaves. The first uses strings of vertex labels, decoded via an algorithm called the Blob Code. The second specifies sets of leaves and interior vertices in spanning trees; the tree a chromosome represents is identified by forming a minimum spanning tree on the interior vertices the chromosome lists and connecting each leaf to the nearest interior vertex, as in the greedy heuristic. Many other codings can represent spanning trees (Raidl and Julstrom, 2003; Rothlauf, 2002, pp.119–197), but in general they do not easily honor the leaf constraint. Section 4 describes the two codings used here, crossover and mutation operators for them, and the sizes of the spaces they represent.

The two codings and operators appropriate for them are implemented in straightforward generational genetic algorithms, which Section 5 describes. The GAs are compared with the greedy heuristic and each other on 65 instances of the leaf-constrained minimum spanning tree problem, 45 Euclidean instances and 20 whose edge weights are chosen randomly. Section 6 describes these comparisons. The greedy heuristic is relatively effective and the Blob-Coded GA does not do as well as the greedy heuristic. The subset-coded GA generally identifies the shortest leaf-constrained minimum spanning trees, though it takes more time than the greedy heuristic does. The subset-coded GA's advantage is greater on the random than on the Euclidean instances.

2. Problem

Given a connected, weighted, undirected graph G on n vertices, a spanning tree is a subgraph of G that connects G's vertices and contains no cycles, and a minimum spanning tree (MST) is a spanning tree on G of minimum total weight. The familiar algorithms of Borůvka (1926; Nešetřil *et al.*, 2001), Kruskal (1956), and Prim (1957) find MSTs on G in times that are polynomial in n. However, constraints on the spanning trees often render the search for a tree of smallest weight NP-hard. Such constraints include an upper bound on trees' degrees (Knowles and Corne, 2000; Narula and Ho, 1980; Raidl, 2000), an upper bound on trees' diameters (Achuthan *et al.*, 1994; Deo and Abdalla, 2000; Julstrom and Raidl, 2003), and the restriction of trees' leaves to exactly two, so that valid trees are Hamiltonian paths.

A leaf of a tree is a vertex in it whose degree is one; that is, the tree connects a leaf to exactly one other vertex. Imposing a lower bound on the number of leaves in spanning trees also makes the search for a tree of lowest weight computationally difficult.

Let ℓ be an integer, $2 \leq \ell < n-1$. A leaf-constrained spanning tree (LCST) is a spanning tree on G that has at least ℓ leaves, and a leaf-constrained minimum spanning tree (LCMST) is an LCST of minimum total weight. The search for an LCMST with at least ℓ leaves on a graph G is the leaf-constrained minimum spanning tree problem. Deo and Micikevicius (1999) showed that it is NP-hard.

Like other problems that seek constrained minimum spanning trees, the LCMST problem has applications to facilities location and to circuit and network design. It is closely related to the p-median problem, in which, given n sites, we seek p locations among them, called medians, so as to minimize the sum of the distances from each site to the nearest median. Indeed, Hoelting *et al.* (1995) describe the LCMST problem as a variation of the p-median problem in which the medians are chosen from among the sites and are themselves connected.

A graph G may have several unconstrained MSTs with a variety of numbers of leaves. If one of these trees has at least ℓ leaves, then it solves the problem, and it can be found quickly. Usually, however, ℓ is a large fraction of the number n of G's vertices ($\ell = 0.9n$, say) and larger than the number of leaves in any unconstrained MST on G. The weight of a leaf-constrained minimum spanning tree is non-decreasing as the number of leaves increases, so the weight of an LCMST with at least ℓ leaves is generally greater than that of an unconstrained MST on G. Figure 1 shows an unconstrained minimum spanning tree on $n = 20$ points in the unit square and a low-weight leaf-constrained spanning tree on the same points with $\ell = 17$ leaves. The MST has weight 3.178 and eight leaves; the LCST has weight 5.041.

Since the LCMST problem is NP-hard, it is a suitable target for heuristics, including evolutionary algorithms. The following sections describe a greedy heuristic and two genetic algorithms for the problem.

3. Greedy Heuristic

Deo and Micikevicius (1999) described a greedy heuristic for the LCMST problem that begins with an unconstrained minimum spanning tree on the target graph G. The algorithm repeatedly exchanges a tree edge and a non-tree edge so as to (a) maintain the structure as a tree; (b) increase the number of leaves in the tree; and (c) increase the tree's weight as little as possible. These iterations stop when either the number of leaves reaches ℓ, in which case

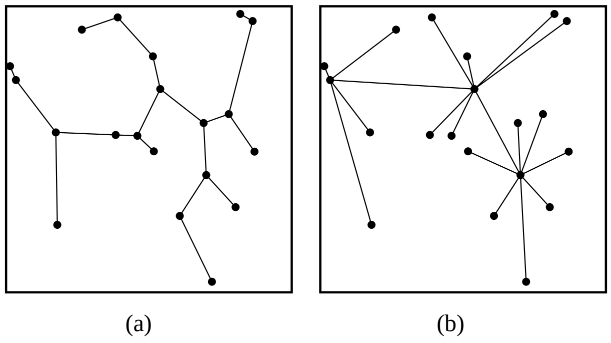


Fig. 1. On a complete graph whose vertices are points in the plane and whose edge weights are the Euclidean distances between the points: (a) An unconstrained minimum spanning tree; it has eight leaves and its weight is 3.178; (b) A low-weight leaf-constrained spanning tree with $\ell = 17$ leaves; its weight is 5.041.

the algorithm returns the spanning tree with ℓ leaves, or no edge-swap is possible that will increase the number of leaves, in which case the algorithm fails to find a tree that satisfies the problem's requirements. The algorithm's time is $O(n^4)$.

A more recent greedy heuristic (Julstrom, 2004) also begins with an unconstrained MST on G but focuses on the target graph's vertices rather than its edges. This heuristic, which we can call ML as in "more leaves," identifies the MST's leaves and interior vertices, then repeats the following step.

ML relabels each interior vertex in turn as a leaf; for each of these vertices, it forms an unconstrained minimum spanning tree on the remaining interior vertices, then connects each leaf, including the new one, to the nearest interior vertex. The new leaf for which the resulting tree is of lowest weight becomes a leaf permanently. ML repeats this step until the number of leaves reaches ℓ. If the target graph G is complete, ML cannot get stuck; it will always return a spanning tree on G with ℓ leaves. Figure 2 presents a pseudo-code sketch of the ML heuristic.

The number of leaf-increasing steps is bounded above by the number n of vertices. Within each step, the number of interior vertices is also less than n, and for each candidate interior vertex, the time required to construct an MST on the interior vertices and attach each leaf to the nearest one is $O(n^2)$. Thus the time of the ML algorithm, like that of the heuristic of Deo and Micikevicius, is $O(n^4)$.

In comparisons using the Euclidean LCMST instances described in Section 6, when $\ell = 0.6n$, ML identified lower-weight trees than the heuristic of Deo and Micikevicius did. When $\ell = 0.9n$, the earlier algorithm always got stuck well below the leaf bound (Julstrom, 2004). It is, then, the ML heuristic to which we compare the performances of the two genetic algorithms that the following sections describe.

```
$T \leftarrow$ MST($V$);
while $|$leaves($T$)$| < \ell$
    $L \leftarrow$ leaves($T$);
    len $\leftarrow$ INFINITY;
    for each $v \in V - L$
        $L_1 \leftarrow L \cup \{v\}$;
        $T_1 \leftarrow$ MST($V - L_1$);
        for each $w \in L_1$
            $u \leftarrow$ vertex in $V - L_1$ nearest $w$;
            $T_1 \leftarrow T_1 \cup \{(w,u)\}$;
        if weight($T_1$) $<$ len
            len $\leftarrow$ weight($T_1$);
            $T \leftarrow T_1$;
report $T$ and weight($T$);
```

Fig. 2. Sketch of the MST-based ML heuristic for the LCMST problem. V is the set of vertices in the target graph G, T is the developing tree, and L is the set of leaves in T.

4. Two Evolutionary Codings

The introduction suggested the usefulness of evolutionary codings that represent only valid candidate solutions to problems of constrained optimization. Edelson and Gargano (2000) presented a genetic algorithm for the leaf-constrained minimum spanning tree problem that used such a coding; it represented spanning trees with at least ℓ leaves on n vertices as strings of $3n - \ell - 2$ symbols. This section describes two more-parsimonious codings of valid LCSTs and operators appropriate for them. The first is based on Prüfer strings; the second specifies interior vertices.

4.1. Blob Code

Cayley's Formula (Cayley, 1889; Even, 1973, pp.103–4) tells us that the number of unconstrained spanning trees in a complete graph on n vertices is n^{n-2}. Prüfer (1918; Even, 1973, pp.104–106) presented a proof of Cayley's Formula in the form of inverse one-to-one mappings between the spanning trees on n vertices and the strings of length $n - 2$ over n vertex labels. We call the strings Prüfer strings, and when we use them to represent spanning trees via Prüfer's mappings, Prüfer numbers.

As representations of spanning trees, Prüfer numbers are deceptively appealing. Each string corresponds to exactly one spanning tree, and we can apply conventional evolutionary operators like k-point crossover and

```
blob ← $\{1, 2, \ldots, n-1\}$;
$T \leftarrow \{(\text{blob} \rightarrow 0)\}$;
for $i$ from 1 to $n-2$
    blob ← blob $-\{i\}$;
    if **path**($b_i$)
        $T \leftarrow T \cup \{(i \rightarrow b_i)\}$;
    else
        $T \leftarrow T \cup \{(i \rightarrow$ **succ**(blob))};
        $T \leftarrow T - \{$(blob → **succ**(blob))};
        $T \leftarrow T \cup \{(\text{blob} \rightarrow b_i)\}$;
blob ← $(n-1)$ in all edges;
```

Fig. 3. Blob decoding algorithm, which makes the edges in the spanning tree a Prüfer string represents explicit. The vertices are labeled $0, 1, \ldots, n-1$. $b_1 b_2 \ldots b_{n-2}$ is a Prüfer string and T is the spanning tree.

position-by-position mutation to these strings. Unfortunately, Prüfer numbers have poor properties for evolutionary search (Gottlieb *et al.*, 2001; Palmer and Kershenbaum, 1994; Rothlauf and Goldberg, 1999). A small change in a Prüfer number generally changes many edges in the tree it represents, and when crossover generates a Prüfer number, the tree the offspring represents generally bears little resemblance to the trees of its parents.

However, there are many other one-to-one mappings between Prüfer strings and spanning trees. Picciotto (1999) examined three of them, which she called the Blob Code, the Dandelion Code, and the Happy Code. Under these mappings, Prüfer strings represent directed trees rooted at vertex 0; we ignore edges' directions to obtain undirected spanning trees. Prüfer strings decoded with the Blob Code have much better characteristics for evolutionary search than Prüfer numbers do (Julstrom, 2001).

Further, strings decoded via the Blob Code (and the other mappings) share with Prüfer numbers a useful property. The degree of each vertex in a spanning tree is always one greater than the number of times its label appears in the string that represents the tree. The labels of a tree's leaves do not appear in the tree's string. Thus we encode spanning trees on n vertices that have at least ℓ leaves as Prüfer strings that contain no more than $n-\ell$ distinct symbols, and we decode these constrained strings with the Blob algorithm.

In the Blob decoding algorithm, the eponymous blob is a set of the graph's vertices. Initially the blob includes all the vertices except vertex 0. Each step removes a vertex from the blob (in ascending order) and records a directed edge in the spanning tree.

The algorithm uses two functions: **succ(v)** returns the successor of vertex v among the edges recorded in the spanning tree so far; and **path(v)** returns TRUE if the directed path from vertex v toward vertex 0 along the tree's edges intersects the blob, FALSE otherwise. Figure 3 summarizes the Blob decoding algorithm. In it, $b_1 b_2 \ldots b_{n-2}$ is a Prüfer string and T is the spanning tree.

Consider identifying the spanning tree on ten vertices $\{0, 1, 2, \ldots, 9\}$ that the string (6 2 7 6 6 2 2 6) represents via the Blob mapping. Initially, the blob contains the vertices 1 through 9 (all except 0), and the tree contains the one edge (blob $\rightarrow$ 0). The first iteration of the algorithm's loop removes vertex 1 from the blob, finds that **path**(6) is TRUE (since vertex 6 is in the blob) and so adds the edge (1 $\rightarrow$ 6) to the tree.

The second loop iteration removes vertex 2 from the blob and finds that **path**(2) is FALSE; no edge yet leads from vertex 2, so the (trivial) path from it does not intersect the blob. In consequence, the algorithm replaces the edge (blob $\rightarrow$ 0) in T with the two edges (2 $\rightarrow$ 0) and (blob $\rightarrow$ 2). Continuing in this way, the algorithm constructs the directed spanning tree that the string represents. Ignoring the edges' directions yields the undirected spanning tree in Fig. 4.

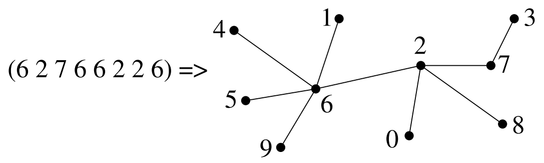


Fig. 4. A Prüfer string of length eight and the spanning tree on ten vertices that it represents via the Blob algorithm.

In the worst case, the Blob algorithm's time is $O(n^2)$, but on average it appears to be no worse than $O(n \log n)$. To implement the algorithm efficiently, represent the tree by an array, say `t[]`, in which the directed edge ($i \rightarrow j$) is indicated by setting `t[`i`]` to j, and let vertex number $n-1$ represent the blob.

4.2. Operators for the Blob Code

Variations of conventional positional operators always yield Prüfer strings that contain no more than $n-\ell$ distinct symbols; that is, ones that represent trees with at least ℓ leaves.

An extension of two-point crossover maintains a set S_{cr} of $n-\ell$ or fewer symbols found in its two parent chromosomes. The operator initializes S_{cr} with the symbols common to both parents. It chooses two cut-points at random within the chromosomes and copies symbols into the offspring from one parent to a cut-point, from the second parent to the second cut-point, then from the first parent again. As it scans the parents and the offspring, it records in S_{cr} the symbols that it encounters, until the set holds $n-\ell$ symbols. After S_{cr} becomes "full", any parental symbol not found in it is not copied into the offspring; the operator writes in its position a random symbol from S_{cr} instead. This mechanism guarantees that any position that contains the same symbol in both parents is preserved and that no more than $n-\ell$ distinct symbols appear in the offspring. Each instance of crossover begins scanning the parents and offspring at a random position and wraps around the chromosomes' ends, so that the operator does not on average place non-parental symbols in some positions more often than in others.

Position-by-position mutation is augmented in the same way. It initializes its set S_{mu} with the symbols of the parent string, and when it replaces a symbol, the new symbol is random if S_{mu} contains fewer symbols than $n-\ell$, and is selected from S_{mu} otherwise. It adds to S_{mu} symbols that are new to the offspring, and removes from S_{mu} symbols that disappear from the offspring. Mutation also begins at a random position in the string each time it is called.

The sets S_{cr} and S_{mu} can be implemented so that every operation on them requires only constant time. Thus the times of the augmented crossover and mutation operators are still $O(n)$.

4.3. Blob Code's Search Space

Constrained Prüfer strings decoded via the Blob Code represent only spanning trees with at least ℓ leaves, and the correspondence between the strings and the trees is one-to-one, so the number of valid strings is the size of the space that a GA using this coding searches. Given n and ℓ, we find this number $T(n,\ell)$.

Let Str(n,s) be the number of Prüfer strings over an alphabet of size n in which *exactly* s symbols appear. Str(n,s) is given by this recurrence:

$$\text{Str}(n,s)=\begin{cases} n & \text{if } s=1,\\ \binom{n}{s}\Big(s^{n-2}-\sum_{q=1}^{s-1}\text{Str}(s,q)\Big) & \text{if } s>1.\end{cases}$$

Clearly, n strings repeat one symbol $n-2$ times. To build a string that contains exactly $s>1$ distinct symbols, choose the symbols, note that each position may be occupied by any one of them, and prohibit all the strings containing fewer than s symbols.

Str(n,s) can also be evaluated using Stirling numbers of the second kind $S_m^{(k)}$ (Even, 1973, pp.60–1), which give the number of ways m items can be partitioned into k non-empty subsets. For $1\le k\le m$, $S_m^{(k)}$ is defined by this recurrence:

$$S_m^{(k)}=\begin{cases} 1 & \text{if } k=1 \text{ or } k=m,\\ S_{m-1}^{(k-1)}+k\,S_{m-1}^{(k)} & \text{otherwise.}\end{cases}$$

Build a string of $n-2$ vertex labels that contains exactly s distinct symbols by choosing the symbols, ordering them, and assigning them in order to non-empty groups of positions in the string, so that

$$\text{Str}(n,s)=\binom{n}{s}s!\,S_{n-2}^{(s)}.$$

However we arrive at Str(n,s), the total number of Prüfer strings in which the number of distinct symbols does not exceed $n-\ell$—thus the number of spanning trees on n vertices that have at least ℓ leaves—is the sum of Str(n,s) for values of s from 1 to $n-\ell$:

$$T(n,\ell)=\sum_{s=1}^{n-\ell}\text{Str}(n,s)).$$

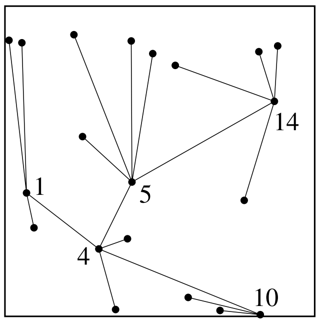
For example, the number of distinct spanning trees on $n=50$ vertices is 50^{48} or about 3.6e81. The number of those that have at least 45 leaves is $T(50,45)\approx$ 7.5e39.

4.4. Subset Coding

As Section 2 observed, ℓ usually exceeds the number of leaves in any unconstrained minimum spanning tree on G, and the weight of an LCMST is non-decreasing as the required number of leaves grows. Thus among the lowest-weight trees with at least ℓ leaves there will be at least one with *exactly* ℓ leaves.

A coding in which each chromosome distinguishes the vertices that must be leaves from those that are not so restricted—interior vertices—limits the search space to a relatively small subset of the spanning trees with at least ℓ leaves. Two steps identify the tree the chromosome represents. The first step builds a minimum spanning tree on the interior vertices; the second connects each leaf to its nearest interior vertex. Note that the resulting tree has minimum weight among all the spanning trees with the same leaves and interior vertices. If another such tree has lower weight, then either a leaf connects to an interior vertex in it by an edge of lower weight, or the spanning tree that connects its interior vertices has lower weight, both contradictions. Note, too, that a vertex that the chromosome specifies to be an interior vertex may be a leaf in the chromosome's tree, if no specified leaf is closer to it than to any other interior vertex.

This coding can be implemented in two ways. A chromosome may be a list of interior vertices or a bit string that distinguishes interior vertices from leaves. Figure 5 shows an example of each implementation and the spanning tree on $n=20$ vertices that both represent. The implementations are equivalent, and each may be transformed into the other in time that is $O(n)$.



(1 4 4 10 14)
(01001100001000100000)

Fig. 5. List of interior vertices, a bit-string that distinguishes leaves and interior vertices, and the spanning tree on 20 vertices that they both represent. The chromosomes indicate that vertices 1, 4, 5, 10, and 14 are interior vertices.

Because this coding represents sets of fixed size—sets of $n - \ell$ vertices—it has been called the fixed-length subset coding. This coding is appropriate when, as here, candidate solutions are, or can be unambiguously derived from, subsets of the problem's elements. It has been used in evolutionary algorithms for outlier detection (Crawford *et al.*, 1995), a problem in chemometrics (Lucasius and Kateman, 1992), and particularly the p-median problem (Alp *et al.*, 2003; Correa *et al.*, 2001; Dibble and Densham, 1993; Estivill-Castro and Torres-Velásquez, 1999; Hoelting *et al.*, 1995; Hosage and Goodchild, 1986; Lim and Xu, 2003).

In an evolutionary algorithm for the LCMST problem, a chromosome's fitness is the total weight of the spanning tree it represents. Prim's algorithm requires time that is $O((n - \ell)^2)$ to find a minimum spanning tree on the interior vertices a chromosome specifies. Identifying the interior vertex nearest each leaf is $O(\ell n)$. This last step can be efficiently implemented when the underlying graph is represented by adjacency lists, each sorted by the weights of the edges in it. Then the interior vertex nearest a leaf will be the first one on the leaf's list, and each search for a nearest interior vertex will examine an average of $n/(n - \ell + 1)$ list entries. The entire time for evaluation remains $O(n^2)$, trough.

4.5. Operators for the Subset Coding

Radcliffe (1993), Radcliffe and George (1993), and Crawford *et al.* (1997) described and analyzed crossover operators for fixed-length subsets. The crossover used here is named (by Radcliffe) the Random Respectful Recombination, abbreviated RRR or R^3. It copies into its one offspring the interior vertices common to both parents, then chooses vertices at random from the remaining parental interior vertices, until the offspring contains $n - \ell$ of them. For example, if $n = 20$ and two parent chromosomes are (1 4 5 10 14) and (2 5 8 12 14), then their offspring might be (1 5 8 10 14).

Mutation simply exchanges a leaf and an interior vertex. If the subsets are implemented as lists of elements, as in the example above, the operations' times are $O(n - \ell)$. If bit-strings implement subsets, the operations' times are simply $O(n)$.

4.6. Subset Coding's Search Space

Under the subset coding, a chromosome specifies a set of $n - \ell$ interior vertices, on which the decoding algorithm identifies a spanning tree of lowest weight, and a complementary set of ℓ leaves. The total number of spanning trees with a particular set of interior vertices is the product of the number of spanning trees on those vertices and the number of ways the leaves can be connected to them:

$$(n - \ell)^{n-\ell-2} \times (n - \ell)^{\ell} = (n - \ell)^{n-2}.$$

Of all of these trees, the subset code represents exactly one, and it has minimum weight among them, as Section 4.4 pointed out. Indeed, the size of the search space when fixed-length subsets represent candidate LCSTs is simply the number of sets of ℓ leaves, or $(n - \ell)$ interior vertices, that can be chosen from the n vertices:

$$\binom{n}{\ell} = \binom{n}{n-\ell}.$$

This number is far smaller than $T(n, \ell)$. For example, while the number of spanning trees on $n = 50$ vertices with at least $\ell = 45$ leaves is $T(n, \ell) \approx 7.5e39$, the number of sets of 45 leaves that can be chosen from 50 vertices is $\binom{50}{45} = 2,118,760$. Note that this value is smaller than the total number of spanning trees with exactly 45 leaves by a factor of

$$(n - \ell)^{n-2} = 5^{48} \approx 3.6e33.$$

Since the subset coding's search space is so much smaller, we would expect a GA using the subset coding to perform decisively better than one using the Blob Code.

5. Two Genetic Algorithms

Two generational genetic algorithms for the LCMST problem encode candidate spanning trees as constrained Prüfer strings decoded via the Blob Code and as fixed-length subsets implemented as bit strings of length n, respectively.

The GAs' initial populations consist of random chromosomes of the appropriate kind. They select chromosomes to participate in crossover and mutation in k-tournaments: k chromosomes are chosen at random from the population and the one that is most fit—that is, the one that represents the leaf-constrained tree of the lowest weight—is selected to be a parent. They apply crossover and mutation separately; each new chromosome is generated by one operator or the other, never both. The probabilities with which the operators are applied are parameters of the algorithms. The GAs are 1-elitist; the single most fit chromosome of each generation is preserved unchanged into the next one. They run through fixed numbers of generations.

Except for their population sizes, the parameters of the two genetic algorithms are the same. On an LCMST problem instance of n vertices, the population of the Blob-Coded GA contains $5n$ chromosomes. Since its search space is so much smaller, the population of the subset-coded GA contains only $2n$ chromosomes. Both algorithms select chromosomes to be parents in tournaments of size two; note that these tournaments assign to chromosomes the same probabilities as linear normalization does (Goldberg and Deb, 1991; Julstrom, 1999). The GAs' probability of crossover is 70% (and their probability of mutation is therefore 30%). In the Blob-Coded GA, the probability that mutation modifies any one symbol in a chromosome is $2/n$. Both algorithms run through $10n$ generations, then report the fitness of the single best chromosome; that is, the total weight of the leaf-constrained spanning tree that the chromosome represents.

6. Comparisons

The greedy ML heuristic, the Blob-Coded GA, and the subset-coded GA were compared on 65 instances of the leaf-constrained minimum spanning tree problem. Forty-five of these instances are Euclidean, fifteen instances each of $n = 50$, 100, and 250 vertices. The remaining twenty instances are non-Euclidean, ten instances each of $n = 100$ and 300 vertices. In every case, the leaf bound ℓ was set to $0.9n$; thus $\ell = 45$, 90, and 225 respectively for the Euclidean instances and $\ell = 90$ and 270 for the non-Euclidean instances.

The Euclidean instances are listed in Beasley's (1990) OR-Library[1] as instances of the Euclidean Steiner problem; they consist of random points in the unit square. We treat the points as the vertices of complete graphs whose edge weights are the distances between the points. The edge weights of the remaining instances were chosen at random on the interval $[0.01, 0.99]$.

[1]`http://mscmga.ms.ic.ac.uk/info.html`

ML was run once and each GA was run 30 independent times on each instance. Tables 1 through 4 summarize the results of these trials. For each instance, the tables list its number, the weight of and number of leaves in an unconstrained minimum spanning tree on its vertices, and the weight of the tree found by the ML heuristic. For each GA applied to each instance, they list the weight of the best tree found in the 30 trials, the mean of the trials' 30 weights, and the standard deviation of those values.

Three observations stand out: (a) The Blob-Coded GA cannot compete with the ML heuristic; (b) the subset-coded GA can; and (c) both the disadvantage of the Blob-Coded GA and the advantage of the subset-coded GA are larger on the random-weight instances than on the Euclidean instances.

Consider first the Blob-Coded GA on the Euclidean instances. Even on the smallest of these, the weights of the GA's best trees exceed the weights of ML's trees by 1.9% to 11.2%, and the average weights of the GA's trees exceed the weights of ML's trees by 14.5% to 28.8%. The GA's disadvantage grows on the larger instances. When $n = 250$ and $\ell = 225$, the weights of the GA's trees are on average nearly 50% greater than the weights of ML's trees.

On the random-weight instances, the Blob-Coded GA's performance does not improve. Here, when $n = 100$ and $\ell = 90$, the weights of the GA's best trees are from 16.1% to 41.2% greater than the weights of ML's trees, and the average weights of the GA's trees are 35.2% to 71.0% greater. On the instances with $n = 300$ and $\ell = 270$, the weights of the Blob-Coded GA's trees are on average 49.7% greater than the weights of the trees ML identifies.

The subset-coded GA does much better. Again, consider the Euclidean instances first. On every instance but one (the last with $n = 50$, a tie), the best tree in each set of 30 trials has smaller weight than the tree ML returns, though the GA's advantage diminishes as the instances get larger.

When $n = 50$ and $\ell = 45$, the weights of the GA's best trees are up to 5.6% smaller than those of ML's trees, with an average improvement over ML of 2.7%. Over all the trials on these instances, the GA's trees have weights on average 2.6% smaller than the weights of ML's trees. When $n = 100$ and $\ell = 90$, the GA's best trees have weights from 2.6% to 4.6% smaller than the weights of ML's trees; over all the trials on these instances, the GA's trees have weights on average 2.4% less than ML's trees. When $n = 250$ and $\ell = 225$, the GA's best trees have weights from 1.5% to 3.6% smaller than the weights of ML's trees, and the GA's trees have weights on average 2.2% smaller than ML's trees.

On the random instances, the subset-coded GA does quite well. Here, when $n = 100$ and $\ell = 90$, the GA and ML tie once, but the best tree the GA finds on the second instance has weight 15.4% smaller than that of ML's tree. On average, the weights of the GA's trees are 7.1% smaller than the weights of ML's trees.

When $n = 300$ and $\ell = 270$, the GA's advantage diminishes slightly. Its best trees are from 4.7% to 10.1% better than ML's trees, and on average the GA's trees have weights 6.0% smaller than those ML identifies.

It is clear that the smaller size of its search space and the effectiveness of its search benefit the subset-coded GA relative to the Blob-Coded GA, but why should its advantage over the ML heuristic be larger on the random than on the Euclidean instances? The structure and regularity of the Euclidean instances may make them amenable to the ML heuristic, whose results can then be only a little improved, while on the random instances there is more room for the subset-coded GA to identify leaf-constrained trees of lower weight.

Figure 6 illustrates the performance of all three algorithms on the first 250-point Euclidean instance. It shows an unconstrained minimum spanning tree on the points and the leaf-constrained spanning trees with 225 leaves identified by the ML heuristic, the Blob-Coded GA, and the subset-coded GA. The unconstrained tree has weight 10.605 and 56 leaves; the ML heuristic's tree has weight 20.655; the Blob-Coded GA's tree has weight 26.389; and the subset-coded GA's tree has weight 20.411.

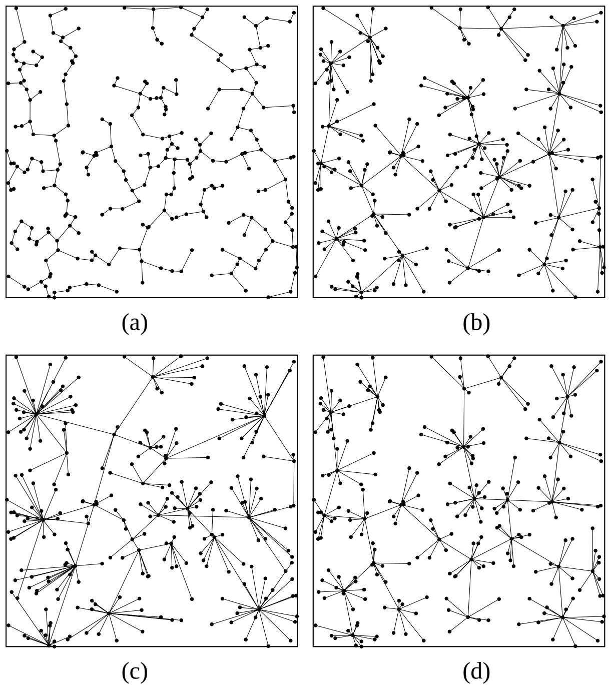


Fig. 6. For the first Euclidean LCMST problem instance with $n = 250$ vertices and $\ell = 225$ leaves: (a) An unconstrained minimum spanning tree on the vertices; it has weight 10.605 and 56 leaves; (b) The spanning tree found by the greedy ML heuristic; it has weight 20.655; (c) The lowest-weight spanning tree found by the Blob-Coded GA; it has weight 26.389; (d) The lowest-weight spanning tree found by the subset-coded GA; it has weight 19.921.

7. Conclusion

Two genetic algorithms seek good solutions to the leaf-constrained minimum spanning tree problem. One encodes spanning trees with at least ℓ leaves as Prüfer strings in which no more than $n - \ell$ distinct symbols appear, and it identifies the trees that these strings represent with the Blob Code mapping. The second encodes a tiny subset of the spanning trees with exactly ℓ leaves as subsets of length $n - \ell$; each specifies the leaves and the interior vertices in a tree.

In tests of 65 instances of the LCMST problem, 45 Euclidean and twenty with randomly chosen weights, the Blob-Coded GA could not compete with a greedy, spanning-tree-based heuristic. The subset-coded GA consistently returned trees of lower weight than did the heuristic, though the GA's advantage was smaller on the larger instances. Its advantage was greater on the random instances, where it identified trees whose weights were as much as 15% less than those of the heuristic's trees. These results support both the primary importance of codings and the operators that act on them in genetic algorithms and the efficacy of searching a smaller space of feasible solutions to the target problem.

The poor performance of the Blob-Coded GA does not necessarily rule out all Prüfer string codings of spanning trees in evolutionary search. Other mappings of Prüfer strings to spanning trees preserve the degree property that made both Prüfer numbers and the Blob Code appear suited to the LCMST problem. These include the Dandelion Code and the Happy Code of Picciotto (1999) and the mapping described by Deo and Micikevicius (2002) from which the represented tree's diameter can be extracted without making the tree itself explicit. Any of these codings, or some other, might be more conducive to evolutionary search than the Prüfer or Blob mappings.

Similarly, the subset-coded GA can be modified in a number of ways that might improve its performance. In particular, Radcliffe (1993), Radcliffe and George (1993), and Crawford *et al.* (1997) described a variety of crossover operators for fixed-length subsets, any of which might provide more effective search, and one can always tinker with a GA's parameters: population size, tournament size, and operator probabilities.

Acknowledgement

My thanks to the reviewers for their cogent and insightful comments.

References

Achuthan N.R., Caccetta L., Caccetta P.A. and Geelan J.F. (1994): *Computational methods for the diameter restricted minimum weight spanning tree problem*. — Australasian J. Combinat., Vol. 10, pp. 51–71.

Alp O., Erkut O. and Drezner Z. (2003): *An efficient genetic algorithm for the p-median problem*. — Ann. Opers. Res., Vol. 122, No. 1–4, pp. 21–42.

Beasley J.E. (1990): *OR-library: Distributing test problems by electronic mail*. — J. Oper. Res. Soc., Vol. 41, pp. 1069–1072.

Borůvka O. (1926): *Přıspěvek k řešenı otázky ekonomické stavby elektrovodnıch sıtı*. — Elektronický obzor, Vol. 15, pp. 153–154.

Cayley A. (1889): *A theorem on trees*. — Quart. J. Math., Vol. 23, pp. 376–378.

Correa E.S., Steiner M.T.A., Freitas A.A. and Carnieri C. (2001): *A genetic algorithm for the p-median problem*. — Proc. *Genetic and Evolutionary Computation Conference*, San Francisco, CA, pp. 1268–1275.

Crawford K.D., Hoelting C.J., Wainwright R.L. and Schoenefeld D.A. (1997): *A study of fixed-length subset recombination*, In: Foundations of Genetic Algorithms 4 (R.K. Belew and M.D. Vose, Eds.). — San Francisco, CA: Morgan Kaufmann, Vol. 4, pp. 365–378.

Crawford K.D., Wainwright K.D. and Vasicek D.J. (1995): *Detecting multiple outliers in regression data using genetic algorithms*. — Proc. ACM Symp. *Applied Computing*, New York, pp. 351–356.

Deo N. and Abdalla A. (2000): *Computing a diameter-constrained minimum spanning tree in parallel*, In: Algorithms and Complexity (G. Bongiovanni, G. Gambosi and R. Petreschi, Eds.). — LNCS, Vol. 1767, pp. 17–31, Berlin: Springer.

Deo N. and Micikevicius P. (1999): *A heuristic for a leaf constrained minimum spanning tree problem*. — Congressus Numerantium, Vol. 141, pp. 61–72.

Deo N. and Micikevicius P. (2002): *A new encoding for labeled trees employing a stack and a queue*. — Bull. Inst. Combinat. Its Appl., Vol. 34, pp. 77–85.

Dibble C. and Densham P.J. (1993): *Generating interesting alternatives in GIS and SDSS using genetic algorithms*. — Proc. GIS/LIS Symposium, Minneapolis, MN, pp. 180–189.

Edelson W. and Gargano M.L. (2000): *Feasible encodings for GA solutions of constrained minimal spanning tree problems*. — *Late-Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, Las Vegas, NV, pp. 82–89.

Estivill-Castro V. and Torres-Velásquez R. (1999): *Hybrid genetic algorithm for solving the p-median problem*, In: Simulated Evolution and Learning: Second Asia-Pacific Conference on Simulated Evolution and Learning (B. McKay, X. Yao, C. S. Newton, J.-H. Kim and T. Furuhashi, Eds.). — LNCS, Vol. 1585, pp. 18–25, Berlin: Springer.

Even R. (1973): *Algorithmic Combinatorics*. — New York: Macmillan.

Goldberg D.E. and Deb K. (1991): *A comparative analysis of selection schemes used in genetic algorithms*, In: Foundations of Genetic Algorithms (G.J.E. Rawlins, Ed.). — San Mateo, CA: Morgan Kaufmann, pp. 69–93.

Gottlieb J., Julstrom B.A., Raidl G.R. and Rothlauf F. (2001): *Prüfer numbers: A poor representation of spanning trees for evolutionary search*. — Proc. *Genetic and Evolutionary Computation Conference*, San Francisco, CA, pp. 343–350.

Hoelting C.J., Schoenefeld D.A. and Wainwright R.L. (1995): *Approximation techniques for variations of the p-median problem*. — Proc. ACM Symp. *Applied Computing*, New York, pp. 293–299.

Hosage C.M. and Goodchild M.F. (1986): *Discrete space location-allocation solutions from genetic algorithms*. — Ann. Oper. Res., Vol. 6, pp. 35–46.

Julstrom B.A. (1999): *It's all the same to me: Revisiting rank-based probabilities and tournaments*. — Proc. Congress *Evolutionary Computation, CEC99*, Vol. 2, pp. 1501–1505, Piscataway, NJ: IEEE Press.

Julstrom B.A. (2001): *The Blob Code: A better string coding of spanning trees for evolutionary search*, In: Genetic and Evolutionary Computation Conference Workshop Program, (A.S. Wu, Ed.). — pp. 256–261, San Francisco, CA.

Julstrom B.A. (2004): *Better greedy heuristics for the leaf-constrained minimum spanning tree problem in complete graphs*. — Discr. Appl. Math., (Submitted).

Julstrom B.A. and Raidl G. (2003): *Greedy heuristics and an evolutionary algorithm for the bounded-diameter minimum spanning tree problem*. — Proc. ACM Symp. *Applied Computing*, pp. 747–752, New York, ACM Press.

Knowles J. and Corne D. (2000): *A new evolutionary approach to the degree constrained minimum spanning tree problem*. — IEEE Trans. Evolut. Comput., Vol. 4, No. 2, pp. 125–134.

Kruskal J.B. (1956): *On the shortest spanning subtree of a graph and the traveling salesman problem*. — Proc. Amer. Math. Soc., Vol. 7, No. 1, pp. 48–50.

Lim A. and Xu Z. (2003): *A fixed-length subset genetic algorithm for the p-median problem*, In: Genetic and Evolutionary Computation – GECCO 2003, (E. Cantú-Paz *et al.*, Eds.). — LNCS, Vol. 2724, pp. 1596–1597, Part II, Berlin: Springer.

Lucasius C.B. and Kateman G. (1992): *Towards solving subset selection problems with the aid of the genetic algorithm*, In: Parallel Problem Solving from Nature II, (R. Männer and B. Manderick, Eds.). — Amsterdam: Elsevier.

Narula G. and Ho C.A. (1989): *Degree-constrained minimum spanning trees*. — Comput. Oper. Res., Vol. 7, No. 4, pp. 39–49.

Nešetřil J., Milková E. and Nešetřilová H. (2001): *Otakar Borůvka on minimum spanning tree problem. Translation of both the 1926 papers, comments, history*. — Discr. Math., Vol. 233, No. 1–3, pp. 3–36.

Palmer C.C. and Kershenbaum A. (1994): *Representing trees in genetic algorithms*. — Proc. 1st IEEE Conf. *Evolutionary Computation*, Orlando, FL, Vol. 1, pp. 379–384.

Picciotto S. (1999): *How to encode a tree*. — Ph.D. thesis, University of California, San Diego.

Prim R.C. (1957): *Shortest connection networks and some generalizations*. — Bell Syst. Tech. J., Vol. 36, No. 6, pp. 1389–1401.

Prüfer H. (1918): *Neuer beweis eines satzes über permutationen*. — Arch. Math. Phys., Vol. 27, No. 3, pp. 142–144.

Radcliffe N.J. (1993): *Genetic set recombination*, In: Foundations of Genetic Algorithms 2, (L.D. Whitley, Ed.). — San Mateo, CA: Morgan Kaufmann, pp. 203–219.

Radcliffe N.J. and George F.A.W. (1993): *A study in set recombination*. — Proc. 5th Int. Conf. *Genetic Algorithms*, pp. 23–30, San Mateo, CA: Morgan Kaufmann.

Raidl G.R. (2000): *An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem*. — Proc. Congress *Evolutionary Computation CEC00*, pp. 104–111, Piscataway, NJ: IEEE Press.

Raidl G.R. and Julstrom B.A. (2003): *Edge-sets: An effective evolutionary coding of spanning trees*. — IEEE Trans. Evolut. Comput., Vol. 7, No. 3, pp. 225–239.

Rothlauf F. (2002): *Representations for Genetic and Evolutionary Algorithms*. — Heidelberg: Physica-Verlag.

Rothlauf F. and Goldberg D. (1999): *Tree network design with genetic algorithms – An investigation in the locality of the Pruefernumber encoding*. — *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, Orlando, FL, pp. 238–243.

Table 1. Results of the trials of the ML heuristic and the two genetic algorithms on the fifteen LCMST problem instances with $n = 50$ and $\ell = 45$. For each instance, the table lists its number, the weight and number of leaves in an unconstrained minimum spanning tree, and the weight of the tree the ML heuristic found. For each GA, it lists the weight of the best tree found in 30 trials, the mean of the 30 weights, and the standard deviation of these values.

				Blob-Coded GA			**Subset-coded GA**		
Instance	w(MST)	Leaves	**ML**	Best	Mean	StdDev	Best	Mean	StdDev
1	4.968	12	10.550	10.753	12.618	1.04	10.111	10.116	0.01
2	5.143	13	10.066	10.912	12.252	0.67	9.652	9.652	0.00
3	4.929	12	9.702	10.047	11.352	0.69	9.243	9.252	0.03
4	4.600	14	8.754	9.572	10.529	0.53	8.510	8.514	0.02
5	5.023	12	9.280	9.496	11.144	0.99	9.161	9.161	0.00
6	5.099	11	9.177	9.846	11.043	0.70	9.032	9.032	0.00
7	4.501	13	8.627	9.120	10.976	1.29	8.480	8.568	0.07
8	4.828	12	9.332	9.852	11.274	0.82	9.128	9.132	0.00
9	4.825	12	9.059	9.690	11.034	0.81	8.578	8.587	0.01
10	4.763	11	8.735	9.304	10.223	0.65	8.554	8.570	0.01
11	4.788	12	9.481	9.982	11.480	0.97	9.236	9.279	0.09
12	4.810	13	8.933	9.931	11.292	0.81	8.807	8.807	0.00
13	4.825	11	9.506	10.496	11.816	0.78	9.398	9.411	0.05
14	4.845	12	10.143	10.579	11.611	0.55	9.577	9.591	0.02
15	4.735	13	8.440	9.028	10.873	0.62	8.440	8.440	0.00

Table 2. Results of the trials of the ML heuristic and the two genetic algorithms on the fifteen LCMST problem instances with $n = 100$ and $\ell = 90$, as in Table 1.

				Blob-Coded GA			Subset-coded GA		
Instance	w(MST)	Leaves	**ML**	Best	Mean	StdDev	Best	Mean	StdDev
1	6.609	20	12.995	16.784	18.909	1.32	12.590	12.626	0.04
2	6.833	20	12.966	14.805	17.234	1.03	12.374	12.398	0.02
3	6.763	25	13.140	15.789	18.567	1.58	12.762	12.774	0.02
4	6.798	18	13.307	15.164	17.950	1.40	12.899	12.911	0.01
5	6.903	24	13.661	15.727	17.949	1.27	13.299	13.326	0.06
6	6.694	25	13.099	16.255	19.063	1.35	12.753	12.772	0.03
7	7.277	20	13.614	15.482	18.599	1.50	13.305	13.345	0.04
8	6.631	23	13.361	16.229	18.336	1.53	12.964	13.039	0.10
9	7.165	28	13.419	16.333	18.500	1.43	13.202	13.238	0.03
10	6.954	25	13.515	15.586	18.198	1.59	13.123	13.136	0.02
11	7.031	25	13.774	17.212	18.734	1.13	13.578	13.583	0.01
12	6.855	23	13.303	16.730	18.888	1.37	13.174	13.213	0.03
13	6.683	18	13.073	14.642	17.360	0.99	12.752	12.796	0.06
14	7.137	26	13.504	16.002	18.943	1.45	13.280	13.317	0.03
15	6.383	23	12.672	14.995	16.410	0.76	12.216	12.226	0.01

Table 3. Results of the trials of the ML heuristic and the two genetic algorithms on the fifteen LCMST problem instances with $n = 250$ and $\ell = 225$, as in Tables 1 and 2.

				Blob-Coded GA			Subset-coded GA		
Instance	w(MST)	Leaves	**ML**	Best	Mean	StdDev	Best	Mean	StdDev
1	10.605	56	20.655	26.389	30.708	2.80	19.921	19.994	0.08
2	10.421	58	20.779	26.703	30.743	2.13	20.279	20.342	0.06
3	10.392	57	20.363	26.710	30.628	2.12	19.841	19.897	0.05
4	10.739	58	20.992	27.984	30.846	1.71	20.455	20.540	0.09
5	10.610	58	20.678	26.203	31.271	2.41	19.957	20.021	0.08
6	10.538	47	20.656	27.580	30.652	1.77	20.341	20.376	0.03
7	10.427	54	20.574	27.520	31.407	2.35	20.025	20.059	0.03
8	10.689	62	20.011	26.976	30.468	1.87	19.594	19.640	0.03
9	10.640	59	21.124	26.536	31.172	3.12	20.461	20.510	0.05
10	10.608	63	20.391	28.535	30.536	1.53	19.888	19.994	0.04
11	10.222	64	19.502	25.328	28.704	1.69	19.044	19.086	0.03
12	10.858	59	20.142	26.722	30.666	2.21	19.844	19.890	0.04
13	10.498	58	20.218	25.223	29.687	2.82	19.777	19.854	0.05
14	10.586	60	20.396	25.743	30.543	2.05	19.938	20.036	0.07
15	10.484	57	20.078	26.181	30.372	2.44	19.494	19.532	0.03

Table 4. Results of the trials of the ML heuristic and the two genetic algorithms on the twenty LCMST problem instances with random distances in $[0.01, 0.99]$. There are ten instances with $n = 100$ and $\ell = 90$ and ten with $n = 300$ and $\ell = 270$. Presentation is as in Tables 1, 2, and 3.

				Blob-Coded GA			**Subset-coded GA**		
Instance	w(MST)	Leaves	**ML**	Best	Mean	StdDev	Best	Mean	StdDev
$n = 100, \ell = 90$									
1	2.190	39	6.174	7.925	9.340	0.84	5.594	5.843	0.14
2	2.047	40	6.588	7.904	9.402	0.82	5.574	5.696	0.09
3	2.332	40	6.219	7.521	9.357	0.94	5.923	5.990	0.08
4	2.029	40	6.343	7.799	8.953	0.75	5.389	5.575	0.17
5	2.214	35	6.850	8.136	9.581	0.74	5.904	5.958	0.09
6	2.311	42	6.952	8.070	9.399	0.81	6.292	6.414	0.09
7	2.007	44	5.748	7.491	8.976	0.84	5.558	5.761	0.11
8	2.071	41	5.889	7.768	9.015	0.80	5.423	5.509	0.13
9	2.162	45	5.138	7.254	8.785	0.79	5.138	5.208	0.06
10	2.301	38	6.652	8.258	9.486	0.91	5.709	5.949	0.12
$n = 300, \ell = 270$									
1	4.106	124	8.221	11.823	12.975	0.65	7.835	7.938	0.06
2	4.266	121	8.637	11.846	12.860	0.63	8.091	8.223	0.07
3	4.131	121	8.611	11.729	12.647	0.68	7.893	8.047	0.08
4	4.348	123	8.814	12.281	13.109	0.53	8.061	8.185	0.06
5	4.124	121	8.701	11.938	12.888	0.78	7.818	7.992	0.10
6	4.182	115	8.646	11.975	12.891	0.60	8.092	8.186	0.06
7	4.152	118	8.544	11.976	13.052	0.61	7.974	8.111	0.09
8	4.048	123	8.795	11.817	12.797	0.52	7.972	8.122	0.07
9	4.228	124	8.575	11.630	12.592	0.76	7.752	7.883	0.09
10	4.203	125	8.166	11.700	12.584	0.55	7.684	7.854	0.06