Initialization is Robust in Evolutionary Algorithms that Encode Spanning Trees as Sets of Edges

Bryant A. Julstrom
Department of Computer Science
St. Cloud State University
720 Fourth Avenue South
St. Cloud, MN 56301 USA
julstrom@eeyore.stcloudstate.edu

Günther R. Raidl
Institute of Computer Graphics and Algorithms
Vienna University of Technology
Favoritenstraße 9–11/1861
1040 Vienna, Austria
raidl@ads.tuwien.ac.at

ABSTRACT

Evolutionary algorithms (EAs) that search spaces of spanning trees can encode candidate trees as sets of edges. In this case, edge-sets for an EA's initial population should represent spanning trees chosen with uniform probabilities on the graph that underlies the target problem instance. The generation of random spanning trees is not as simple as it might appear. Mechanisms based on Prim's and Kruskal's minimum spanning tree algorithms are not uniform, and uniform mechanisms are slow, not guaranteed to terminate, or require that the underlying graph be complete.

However, a non-uniform initial population of spanning trees need not harm an EA's performance. Trials of a crossover-only EA for the One-Max-Tree problem, using Prim-based, Kruskal-based, and uniform initialization, indicate that the distribution of *edges* in the initial population is far more important than the distribution of *trees*. A skewed distribution of edges in its initial population damages the EA's performance, but this is remedied by a reasonable amount of mutation.

Keywords

Spanning trees, sets of edges, random spanning trees, initialization

1. INTRODUCTION

A spanning tree of an undirected graph G is a subgraph of G that connects all of G's vertices and contains no cycles. While it is computationally easy to find a minimum-weight spanning tree in a weighted graph, many problems that search spaces of spanning trees are NP-hard. To such problems, we apply heuristics, including evolutionary algorithms (EAs).

Evolutionary algorithms have encoded spanning trees in a variety of ways, including characteristic vectors, predecessor codings, Prüfer numbers, node and link-and-node biasing [Palmer and Kershenbaum, 1994], and network random keys [Rothlauf *et al.*, 2000]. Several authors have recently argued for the direct representation of spanning trees as sets of their edges [Li and Bouchebaba, 1999; Gottlieb, *e*t al., 2001]. This coding can usually be evaluated quickly, and crossover and mutation operators for it offer high heritability and locality.

An EA's initial population usually consists of uniformly distributed random chromosomes, in this case the edge-sets of spanning trees chosen at random on the graph that underlies the problem instance. Unfortunately, the generation of uniformly random spanning trees is not as simple as it might appear. Mechanisms inspired by Prim's [1957] and Kruskal's [1956] minimum spanning tree algorithms do not associate uniform probabilities with the spanning trees of a graph. Uniform algorithms are slow, are not guaranteed to terminate, or require that the underlying graph be complete.

However, a non-uniform initial population of spanning trees need not harm an EA's performance. Tests using a crossover-only EA for the One-Max-Tree problem [Rothlauf *et al.*, 2000] indicate that the distribution of *edges* in the initial population is far more important than the distribution of *trees*. A skewed distribution of edges in the initial population can damage the EA's performance, but this can be remedied by a reasonable amount of mutation.

The following sections of the paper review the problem of generating random spanning trees, describe and analyze spanning tree generators based on Prim's and Kruskal's algorithms, summarize algorithms that associate uniform probabilities with spanning trees, examine the effects of the initialization algorithm and the distribution of edges on the performance of an EA for the One-Max-Tree problem, and note the compensating effect of mutation. We conclude that spanning trees may be chosen for an EA's initial population in any way that ensures broad representation of the underlying graph's edges.

2. RANDOM SPANNING TREES

When an evolutionary algorithm searches a solution space of size N and no special problem-specific knowledge is to be exploited, we generally seek to initialize its population so that the space's elements are equally likely to appear:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

SAC 2002, Madrid, Spain.

©2002 ACM 1-58113-445-2/02/03...$5.00

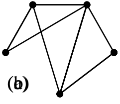

Figure 1: The three structures of spanning trees on five vertices: A path (a), a path of four vertices with a branch (b), and a star (c).

When one will be selected, each has probability $1/N$. The number of spanning trees on a complete undirected graph of n vertices is n^{n-2} [Cayley, 1889; Even, 1973, pp.98–106]. When an EA will search this space, we would like to choose spanning trees for its initial population so that, when one will be chosen, each has probability $1/n^{n-2}$. The following assumes a complete graph and the feasibility of all its spanning trees, but the tendencies we show remain valid in the absence of these conditions.

We analyze "random" spanning tree algorithms by examining the expected numbers of leaves and the probability that a particular vertex is a leaf in trees they produce. For example, in a complete graph on $n = 5$ vertices, a spanning tree may be a path, with two leaves; a path of four vertices with a branch, for three leaves; or a star, with four leaves. Figure 1 illustrates these possibilities. Of the $5^3 = 125$ spanning trees on five vertices, $5!/2 = 60$ are paths, $\binom{5}{4}4!/2 = 60$ are paths with a branch, and five are stars, so the expected number of leaves in a uniformly random spanning tree on five vertices is

$$2 \cdot \frac{60}{125} + 3 \cdot \frac{60}{125} + 4 \cdot \frac{5}{125} = \frac{64}{25} = 2.56,$$

and the probability that a particular vertex is a leaf in such a tree is $2.56/5 = 0.512$.

Rényi [1959] showed that in general the number of leaves in a random spanning tree on n nodes approaches the normal distribution $\mathcal{N}(n/e, n(e-2)/e^2)$ as $n \to \infty$. In particular, the expected number of leaves—the mean of this distribution—approaches n/e, and the probability that a particular vertex will be a leaf therefore approaches $1/e \doteq 0.368$.

3. ADAPTING PRIM'S ALGORITHM

Prim's algorithm [1957] builds a minimum spanning tree on a weighted graph G by starting from an arbitrary vertex and repeatedly appending the cheapest edge between a vertex currently in the tree and one that is not. If the algorithm instead chooses each edge at random from those currently eligible, it returns a "random" spanning tree on the graph. We called the resulting algorithm PrimRST [16]. In the following sketch, V is the set of vertices of G, E is the edge-set of G, T is the set of tree-edges, C is the set of vertices T connects, and v_o, u, and v are vertices.

```
PrimRST(V, E):
v_o ← a random vertex in V;
T ← ∅;
C ← {v_o};
while |C| < n
    u ← a random vertex in C;
    v ← a random vertex in V − C | (u, v) ∈ E;
    T ← T ∪ {(u, v)};
    C ← C ∪ {v};
```

This algorithm does not associate uniform probabilities with the spanning trees of G. In [16] we show that PrimRST assigns higher-than-uniform probabilities to star-like trees and lower-than-uniform probabilities to path-like trees. Here, consider the probability that a particular vertex v_a is a leaf in a tree returned by PrimRST.

Let G be a complete graph on n nodes. Each step of PrimRST selects one vertex to join the growing spanning tree. The first chooses v_o, and the ith step—an iteration of the algorithm's loop—chooses the tree's i^{th} vertex. For v_a to be a leaf, it must be chosen from $V - C$ at some step i, then never chosen from C. Let $p_i(v_a)$ be the probability of this event, and note that the probability that v_a is a leaf is the sum of these values over all PrimRST's steps:

$$P[v_a \text{ is a leaf}] = \sum_{i=1}^{n} p_i(v_a).$$

In general, v_a is chosen at a random step; the probability that it is the ith step ($i = 1, \ldots, n$) is $1/n$.

The probability that v_a is chosen as the start vertex and remains a leaf is

$$p_1(v_a) = \frac{1}{n} \cdot 1 \cdot \frac{1}{2} \cdot \frac{2}{3} \cdot \ldots \cdot \frac{n-2}{n-1} = \frac{1}{n(n-1)}.$$

If v_a joins the tree on step $i > 1$, $|C|$ becomes i and the probability that v_a is never chosen from C during all following steps is

$$\frac{i-1}{i} \cdot \frac{i}{i+1} \cdot \ldots \cdot \frac{n-2}{n-1} = \frac{i-1}{n-1}.$$

It follows that

$$p_i(v_a) = \frac{1}{n} \cdot \frac{i-1}{n-1} = \frac{i-1}{n(n-1)}, \text{ for } i = 2, 3, \ldots, n,$$

and the probability that v_a is a leaf is

$$P[v_a \text{ is a leaf}] = \frac{1}{n(n-1)} + \sum_{i=2}^{n} \frac{i-1}{n(n-1)}$$

$$= \frac{1}{n(n-1)} + \frac{1}{n(n-1)} \sum_{i=1}^{n-1} i$$

$$= \frac{1}{n(n-1)} + \frac{1}{n(n-1)} \frac{n(n-1)}{2} = \frac{1}{n(n-1)} + \frac{1}{2}.$$

The expected number of leaves in a PrimRST spanning tree is then

$$E[\text{leaves}] = n \cdot P[v_a \text{ is a leaf}] = \frac{1}{n-1} + \frac{n}{2}.$$

For example, the probability that a particular vertex will be a leaf when PrimRST generates a spanning tree on $n = 5$ vertices is $1/(5 \cdot 4) + 1/2 = 11/20 = 0.55$, and the expected number of leaves in such a tree is $1/4 + 5/2 = 11/4 = 2.75$.

More generally, as n increases, the probability that a particular vertex will be a leaf approaches $1/2$, and the expected number of leaves approaches $n/2$. These are larger than the corresponding values for uniformly random spanning trees ($1/e$ and n/e), and they confirm the bias toward stars.

4. ADAPTING KRUSKAL'S ALGORITHM

Kruskal's algorithm [1956] builds a minimum spanning tree on a weighted graph G by examining G's edges in order of increasing weight. It includes in the spanning tree the edges that join currently unconnected components. If the algorithm instead examines G's edges in random order, it returns a "random" spanning tree on G. We have called this algorithm KruskalRST [16]; in the following sketch, V and E are the vertices, respectively the edges, of G, T is the set of edges in the spanning tree, and e is an edge.

```
KruskalRST(V, E):
T ← ∅;
while |T| < n − 1
   e ← a random edge in E;
   E ← E − {e};
   if e's vertices are not connected in T
      T ← T ∪ {e};
```

Like PrimRST, this algorithm favors stars and disfavors paths, though not as severely as does PrimRST.

Consider the expected number of leaves and the probability that a particular vertex is a leaf in a tree returned by KruskalRST. Unfortunately, it seems to be impossible to find closed-form expressions for these quantities; the probabilities as each new edge is added to T depend on its current structure, and the number of possible structures is vast even for moderate n. Instead, we pursue an example (which illustrates the difficulties) and confirm empirically the intuitions it suggests.

Again, consider generating a spanning tree on a complete graph with $n = 5$ vertices. Restricting our attention to edges that join components of the developing tree, KruskalRST includes four (in general, $n - 1$) edges to complete a tree. Figure 2 shows the intermediate structures the algorithm might develop and how they lead to the three possible tree structures on five vertices.

Figure 2 also shows the probabilities with which the selection of the next eligible edge transforms one structure into another, and the probabilities that the structures appear. Each structure's probability is a weighted sum of the probabilities of its possible predecessors; the weights are the probabilities that the structure is produced from those predecessors.

For example, a path of three vertices appears with probability 2/3 after two edges have been chosen, and a pair of isolated edges appears with probability 1/3. A path of four vertices arises from the former with probability 4/7 and from the latter with probability 1/2; the probability that the first three edges will form a path is

$$\frac{2}{3} \cdot \frac{4}{7} + \frac{1}{3} \cdot \frac{1}{2} = \frac{23}{42}.$$

The three possible spanning tree structures have these probabilities: for a path, 113/252; for a path with a branch, 127/252; and for a star, 12/252. The expected number of leaves in such a tree is then

$$2 \cdot \frac{113}{252} + 3 \cdot \frac{127}{252} + 4 \cdot \frac{12}{252} = \frac{655}{252} \doteq 2.5992,$$

and the probability that a vertex is a leaf is $2.5992/5 \doteq 0.5198$. Note that these values are less than those of

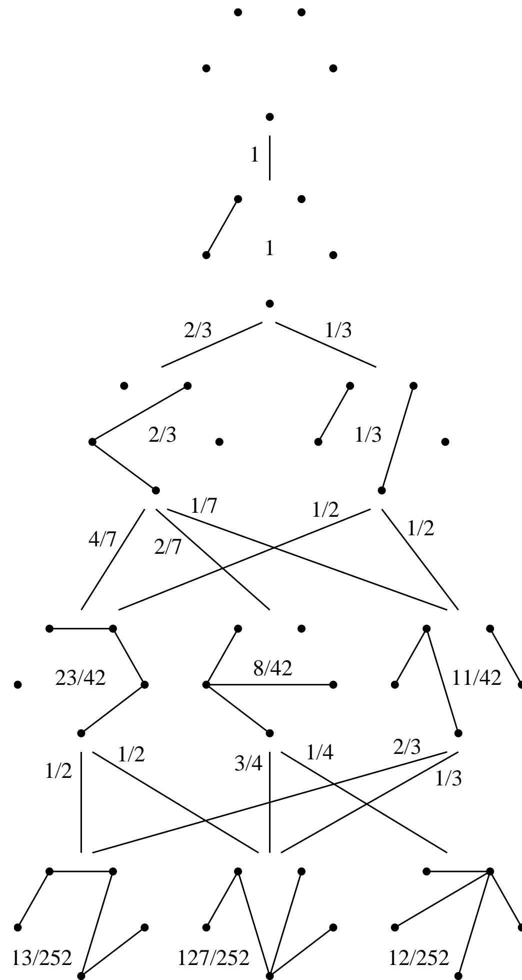

Figure 2: The steps KruskalRST might take in generating a spanning tree on five vertices, the probabilities of the steps, and the probabilities of the resulting structures.

PrimRST on five vertices (2.75 and 0.55, respectively) but more than those of a uniformly random tree (2.56 and 0.512).

Tests confirm the biases that the example suggests. A program used PrimRST, KruskalRST, and a uniform algorithm (decoding Prüfer strings, as the next section describes) to generate spanning trees on complete graphs of 5 to 500 vertices. Each algorithm returned 5 000 trees for each graph size.

Table 1 reports the average numbers of leaves and the empirical probabilities that a particular vertex is a leaf in the sets of trials. Only for uniformly random spanning trees do these values approach the limits identified by Rényi: $E[\text{leaves}] \to n/e$ and $P[v_a \text{ is a leaf}] \to 1/e \doteq 0.368$.

Clearly the average number of leaves in trees generated by PrimRST converges to $n/2$, and the probability that a particular vertex is a leaf to 1/2, conforming to the analysis of Section 3. In the uniformly random trees, the average number of leaves approaches n/e, and the leaf probability approaches $1/e$. The values for KruskalRST always fall between those for PrimRST and the uniform algorithm; the leaf probability for KruskalRST appears to be approaching

Table 1: Empirically observed average numbers of leaves and the probabilities that a particular vertex is a leaf in spanning trees returned by PrimRST, KruskalRST, and a uniform algorithm.

	PrimRST		KruskalRST		Uniform	
n	Leaves	$P[v_a]$	Leaves	$P[v_a]$	Leaves	$P[v_a]$
5	2.738	0.548	2.597	0.519	2.557	0.511
10	5.108	0.511	4.495	0.450	4.301	0.430
15	7.557	0.504	6.483	0.432	6.118	0.408
20	10.071	0.504	8.446	0.422	7.939	0.397
25	12.534	0.501	10.476	0.419	9.749	0.390
30	15.017	0.501	12.469	0.416	11.608	0.387
40	20.063	0.502	16.573	0.414	15.263	0.382
50	24.981	0.500	20.552	0.411	18.915	0.378
60	30.004	0.500	24.628	0.410	22.631	0.377
80	40.011	0.500	32.766	0.410	29.970	0.375
100	50.002	0.500	40.882	0.409	37.324	0.373
150	75.012	0.500	61.210	0.408	55.732	0.372
200	99.953	0.500	81.461	0.407	74.186	0.371
250	125.017	0.500	101.798	0.407	92.537	0.370
300	149.956	0.500	122.130	0.407	110.940	0.370
400	199.967	0.500	162.828	0.407	147.581	0.369
500	249.982	0.500	203.529	0.407	184.450	0.369

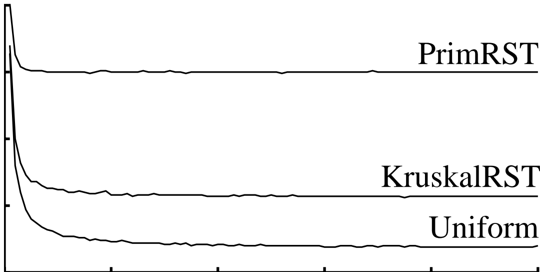

Figure 3: Empirical probabilities $P[v_a]$ that a particular vertex is a leaf in spanning trees on n vertices generated by PrimRST, KruskalRST, and a uniform algorithm, as n grows.

a value near 0.407. KruskalRST is less biased than PrimRST but still favors trees with more leaves. Figure 3 plots the observed probabilities that a particular vertex is a leaf in trees generated by the three algorithms.

5. UNIFORMLY RANDOM ALGORITHMS

While implementations of PrimRST and KruskalRST can be fast, uniformly random algorithms are usually computationally more expensive. Guénoche [1983] described an algorithm that associates uniform probabilities with the spanning trees of an arbitrary graph G. It uses the fact that the number of spanning trees on a graph of n vertices can be identified by computing a determinant of size $n \times n$. The algorithm assumes an ordering of G's edges; this induces a lexicographic order on the set of spanning trees, each of which is represented by the ordered set of its edges. The ith spanning tree can be found by computing at most $m = |E|$ determinants, and the algorithm's time is $O(n^3 m)$. By computing fewer determinants, Colbourne *et al.* [1989] reduced the algorithm's time to $O(n^3)$.

Broder [1989] described an algorithm based on a random walk in G. A particle starts at an arbitrary vertex in G. At each step, it moves over a randomly chosen edge incident to the current vertex. When the particle visits a vertex for the first time, the edge it moved over joins the spanning tree. The algorithm terminates when the particle has visited all of G's vertices. While the algorithm's worst-case time is unbounded, it terminates in $O(n \log n)$ *expected* time for almost all graphs and in time that is $O(n^3)$ for a few special cases. Wilson [1996] presented an improved random-walk algorithm whose expected time is only linear for many graphs.

For a complete graph G on n vertices, Prüfer [1918] described a constructive one-to-one mapping between vectors of $n-2$ vertex labels and spanning trees on the n vertices. These vectors are called Prüfer numbers, and the degree of each vertex in the spanning tree a Prüfer number represents is always one greater than the number of appearances of the vertex's label. The following algorithm uses this property to identify the spanning tree T that a Prüfer number $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_{n-2})$ represents [Even, 1973, pp.104–106]:

> **To decode a Prüfer number α:**
> scan the string to identify the vertices' degrees;
> $T \leftarrow \emptyset$;
> for i from 1 to $n-2$
> $v \leftarrow$ vertex of degree 1 with the smallest label;
> $T \leftarrow T \cup \{(v, \alpha_i)\}$;
>   decrement the degrees of v and α_i;
> two vertices v_1 and v_2 remain with degree 1;
> $T \leftarrow T \cup \{(v_1, v_2)\}$;

For example, when G's vertices are labeled $\{1,2,3,4,5,6\}$, the Prüfer number $(4,5,2,2)$ represents the tree $T = \{(1,4), (3,5), (4,2), (5,2), (2,6)\}$. The algorithm identifies these edges in the listed order.

With a priority queue implemented in a heap to hold the vertices of current degree one, the Prüfer decoding algorithm runs in time that is $O(n \log n)$. Since the generation of a random Prüfer number requires only linear time, the time to generate a random spanning tree via a Prüfer number is also $O(n \log n)$.

There are many other mappings like Prüfer's from vectors of $n-2$ vertex labels (we call them generally Prüfer strings) to spanning trees. Edelson and Gargano [2001] proposed one with a linear time decoding algorithm. Picciotto [1999] and Deo and Micikevicius [2001] have described several others. One, called the Blob Code, exhibits stronger locality and heritability than do Prüfer numbers, and an EA for the One-Max-Tree problem performed significantly better when it encoded spanning trees via the Blob Code than with Prüfer numbers [9].

When the underlying graph G is complete, uniformly random spanning trees are most efficiently generated by decoding random Prüfer numbers via such Prüfer-like mappings. However, the applicability of such a scheme is limited in two ways. First, it is restricted to complete graphs. Second, it cannot easily honor problem-specific constraints on such features as a tree's diameter or edge capacities. For all Prüfer-like mappings, each vertex's degree in the represented spanning tree is one more than the number of times its symbol appears in the vector, so these codings can honor constraints on degrees or numbers of leaves.

Table 2: Results of the EA for the One-Max-Tree problem with random target trees when using Prim-based (P-RST), Kruskal-based (K-RST), uniform, or skewed initialization and only crossover, and with skewed initialization and 30% mutation (w/mut). Each cell shows the mean and the standard deviation of the best solutions' fitnesses.

n	P-RST	K-RST	Uniform	Skewed	w/mut
50	44.48	45.50	45.22	38.96	46.04
	1.58	1.33	1.92	1.88	1.43
100	90.00	90.16	89.72	76.90	91.64
	2.61	2.84	2.31	3.21	2.37
150	133.66	134.30	133.76	118.24	137.30
	3.52	2.81	3.36	3.45	2.43
200	177.40	178.04	177.76	153.26	183.7
	3.60	4.26	3.73	4.34	2.79
250	221.24	221.60	221.26	196.12	229.88
	4.00	3.70	3.89	3.87	3.06
300	264.10	264.88	265.10	228.70	274.48
	4.76	4.65	5.46	4.98	4.09

Table 3: Distributions of edges in 1 000 spanning trees on $n = 8$ vertices generated by the uniform algorithm (a) and by the Skewed algorithm (b).

(a)	1	2	3	4	5	6	7
0	240	249	255	258	262	213	269
1		246	239	264	248	243	254
2			258	255	223	265	248
3				249	240	264	234
4					244	264	257
5						246	270
6							243

(b)	1	2	3	4	5	6	7
0	237	484	256	458	250	457	252
1		269	25	233	24	273	25
2			269	464	252	465	251
3				240	23	245	21
4					241	500	272
5						249	29
6							236

6. COMPARISONS

In the well-known One-Max problem, a bit string's fitness is the number of 1's it contains. The One-Max-Tree problem [Rothlauf *et al.*, 2000] is a similar exercise in the space of spanning trees. Given a connected graph G on n vertices, it specifies a target spanning tree; the fitness of any spanning tree on G is the number of edges it shares with the target.

We compared the initialization schemes of the previous three sections in a generational EA for the One-Max-Tree problem. The EA represents spanning trees as sets of their edges, and it initializes its population using PrimRST, KruskalRST, or a uniform scheme (decoding random Prüfer strings).

The algorithm uses crossover to generate all offspring trees; only edges present in the initial population are available to build new solutions. The EA selects parents for crossover in tournaments of size two without replacement. Kruskal's algorithm inspires the crossover operator: After copying into the offspring all the edges that are common in both parents, it joins the components so created with edges randomly selected from the remaining parental edges. The EA applies standard generational replacement with 1-elitism and runs through a fixed number of generations.

The EA was run 50 times with each initialization operator on instances of One-Max-Tree with uniformly random target trees and numbers of vertices from 50 to 300. On n vertices, its population size was $5n$ and it ran through $2n$ generations.

The first three columns of Table 2 summarize these tests. Each cell displays the mean and standard deviation of the best fitnesses in one set of trials. At every number of vertices, there are no significant differences among the average best fitnesses when the EA initialized its population with PrimRST, KruskalRST, and the uniform algorithm. The choice of initialization operator does not affect the EA's performance on the One-Max-Tree problem.

Why should this be so, when target trees are uniformly random and both KruskalRST and especially PrimRST favor star-like structures? Because the *structures* of the trees in the initial population do not matter. Using crossover alone, the EA builds fitter trees using the edges present in the initial population; a uniform distribution of *edges* is thus most conducive to the EA's progress, and this is what all three initialization algorithms provide. Though PrimRST and KruskalRST favor stars over other trees, none of the three algorithms favors any edges; all provide the same raw material for the evolution of fitter trees.

This suggests that a skewed distribution of edges in the EA's initial population will damage its performance. We tested this conjecture, and the explanation above, with such initial populations. A function called *Skewed* decoded non-uniformly random Prüfer strings in which even vertex numbers were 19 times as likely to appear as odd. Table 3 compares empirical distributions of edges for the uniform algorithm and this one with $n = 8$; in the trees generated by Skewed, edges joining odd-numbered vertices are rare, those joining even-numbered vertices are over-abundant.

The fourth column of Table 2 presents the results of trials using Skewed initialization in the EA for the One-Max-Tree problem. At every number of vertices, the mean fitness using Skewed is less than those for PrimRST, KruskalRST, and the uniform algorithm, and the differences are all significant.

Mutation is usually used to (re-)introduce genetic information into the population, so if the relative failure of the EA using Skewed initialization is due to the unbalanced distribution of edges in the initial population, incorporating mutation into the algorithm should restore its performance.

Position-by-position mutation for sets of edges scans a chromosome and, with a small probability, replaces each edge by a new feasible random edge that reconnects the tree. The fifth column of Table 2 summarizes the performance of the EA with Skewed initialization and this operator. For these trials, the probability of crossover was reduced to 70% and mutation generated the remaining 30% of all offspring. With

n vertices, the probability that any one edge was mutated was $1/n$.

For every number of vertices, the EA with Skewed initialization and mutation performs slightly better than any previous version. Mutation does indeed compensate for the poor distribution of edges in the initial population. More generally, these results suggest that when an evolutionary algorithm encodes candidate spanning trees as sets of edges, the mechanism by which it generates its initial population is unimportant as long as it and the algorithm's mutation operator provide an adequate selection of edges.

7. CONCLUSION

When an evolutionary algorithm that searches a space of spanning trees encodes candidate trees as sets of edges, its initial population consists of "random" spanning trees on the underlying graph G. Mechanisms to generate such trees based on Prim's and Kruskal's minimum spanning tree algorithms do not associate uniform probabilities with the spanning trees of G; both favor stars and disfavor paths. Algorithms that yield uniform spanning trees are slower or not guaranteed to terminate, except for the decoding of random Prüfer strings, which applies only when G is complete.

However, the choice of the initialization algorithm is not as important as it might appear. Trials of a crossover-only EA for the One-Max-Tree problem found no significant differences in performance between Prim-based, Kruskal-based, and uniform initialization, apparently because all three initializations yield uniform distributions of *edges*, the raw material from which the EA constructs fitter trees. This analysis is supported by the relative failure of the EA with an edge-skewed initialization operator and by the restoration of that algorithm's performance by mutation.

These results suggest that when an evolutionary algorithm encodes spanning trees as sets of edges, the mechanism it uses to generate the trees in its initial population is less important than the variety of edges that initialization and mutation together provide.

Acknowledgements

This work is supported by the Austrian Science Fund (FWF) under the grant P13602–INF.

8. REFERENCES

[1] A. Broder. Generating random spanning trees. In *IEEE 30th Annual Symposium on Foundations of Computer Science*, pages 442–447. IEEE, 1989.

[2] A. Cayley. A theorem on trees. *Quarterly Journal of Mathematics*, 23:376–378, 1889.

[3] C. J. Colbourne, R. P. J. Day, and L. D. Nel. Unranking and ranking spanning trees of a graph. *Journal of Algorithms*, 10(2):249–270, 1989.

[4] N. Deo and P. Micikevicius. Comparison of Prüfer-like codes for labeled trees. Baton Rouge, LA, 2001. Thirty-Second Southeastern International Conference on Combinatorics, Graph Theory, and Computing.

[5] W. Edelson and M. Gargano. A modified Prüfer code: $O(n)$ implementation. In *Graph Theory Notes of the N.Y. Academy of Sciences*, pages 37–39. N.Y. Academy of Sciences, 2001.

[6] S. Even. *Algorithmic Combinatorics*. The Macmillan Company, New York, 1973.

[7] J. Gottlieb, B. A. Julstrom, G. R. Raidl, and F. Rothlauf. Prüfer numbers: A poor representation of spanning trees for evolutionary search. In L. Spector et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO–2001*, pages 343–350, San Francisco, CA, 2001. Morgan Kaufmann.

[8] A. Guénoche. Random spanning tree. *Journal of Algorithms*, 2:214–220, 1983.

[9] B. A. Julstrom. The Blob Code: A better string coding of spanning trees for evolutionary search. In Wu [20], pages 256–261. July 7.

[10] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematics Society*, 7(1):48–50, 1956.

[11] Y. Li and Y. Bouchebaba. A new genetic algorithm for the optimal communication spanning tree problem. In C. Fonlupt, J.-K. Hao, E. Lutton, E. Ronald, and M. Schoenauer, editors, *Proceedings of Artificial Evolution: Fifth European Conference*, volume 1829 of *LNCS*, pages 162–173. Springer, 1999.

[12] C. C. Palmer and A. Kershenbaum. Representing trees in genetic algorithms. In D. Schaffer, H.-P. Schwefel, and D. B. Fogel, editors, *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 379–384. IEEE Press, 1994.

[13] S. Picciotto. *How to encode a tree*. PhD thesis, University of California, San Diego, 1999.

[14] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[15] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv für Mathematik und Physik*, 27:142–144, 1918.

[16] G. R. Raidl and B. A. Julstrom. Edge-sets: An effective evolutionary coding of spanning trees. *IEEE Transactions on Evolutionary Computation*, 2001. Submitted.

[17] A. Rényi. Some remarks on the theory of trees. *Magyar Tudományos Akadémia Matematikai Kutató Intézetének Közleményei*, 4:73–85, 1959. Reprinted in *Selected Papers of Alfréd Rényi* (Pál Turán, Ed.), volume 2. Budapest: Akadémiai Kiadó, 1976.

[18] F. Rothlauf, D. Goldberg, and A. Heinzl. Network random keys – a tree network representation scheme for genetic and evolutionary algorithms. Technical Report 2000010, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, 2000.

[19] D. B. Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the Twenty-eighth Annual Symposium on the Theory of Computing*, pages 296–303, N.Y., 1996. ACM Press.

[20] A. S. Wu, editor. *2001 Genetic and Evolutionary Computation Conference Workshop Program*, San Francisco, CA, 2001. July 7.