Some Novel Locality Results for the Blob Code Spanning Tree Representation

Tim Paulden and David K. Smith
University of Exeter
Exeter, United Kingdom
t.j.paulden@exeter.ac.uk, d.k.smith@exeter.ac.uk

ABSTRACT

The Blob Code is a bijective tree code that represents each tree on n labelled vertices as a string of $n-2$ vertex labels. In recent years, several researchers have deployed the Blob Code as a GA representation, and have reported promising results across a range of tree-based optimization problems.

In this paper, we exploit a recently discovered linear-time decoding algorithm for the Blob Code to develop some novel locality results, extending previous work by Julstrom.

Let Δ be the random variable representing the number of tree edges that are changed by a random single-element string mutation. Under the Blob Code, we demonstrate that pessimal mutations (i.e., mutations for which $\Delta = n-1$) can arise for any $n > 4$. However, as n grows, the probability of perfect mutation $P(\Delta = 1)$ approaches one, following a power-law relationship, and $E(\Delta)$ approaches two. These results show that the locality of the Blob Code is high, but not as high as that of Dandelion-like codes.

We also show that the choice of mutation position places restrictions on the range of Δ, and therefore influences the distribution of Δ. In particular, mutating the kth element of a Blob string alters at most $n-k$ tree edges.

Categories and Subject Descriptors

E.1 [**Data Structures**]: *Arrays, Graphs and Networks, Trees*; G.2 [**Discrete Mathematics**]: Applications; G.2 [**Discrete Mathematics**]: Combinatorics—*Combinatorial Algorithms*; G.2 [**Discrete Mathematics**]: Graph Theory—*Trees, Graph Algorithms*; F.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on Discrete Structures*

General Terms

Algorithms, Theory

Keywords

Spanning tree, Representation, Encoding, Coding, Locality

1. INTRODUCTION

1.1 Representing trees in genetic algorithms

There are numerous ways to represent tree structures as linear strings of symbols [13],[19],[20]. However, for such a representation to perform effectively in a genetic algorithm (GA), five key properties [13],[20] must be satisfied:

1. *'Full coverage'* or *'completeness'* — Every tree should be represented by at least one string;
2. *'Zero bias'* — Every tree should be represented by the same number of strings;
3. *'Perfect feasibility'* or *'closure'* — Every string should represent a valid tree;
4. *'Efficiency'* — Encoding (converting tree to string) and decoding (converting string to tree) should be fast;
5. *'High locality'* — Making small changes to a string should lead to small changes in the corresponding tree.

A tree representation is a *Cayley code* [20] if it represents each possible tree on n labelled vertices as a string of $n-2$ vertex labels, such that the correspondence between trees and strings is bijective (i.e., each tree corresponds to a unique string, and each string corresponds to a unique tree).

Cayley codes are an attractive method for representing trees in GAs, as their bijective nature ensures that properties 1 to 3 above are automatically satisfied [14].

Many different Cayley codes have been described in the mathematical literature. These codes divide naturally into two categories: *Prüfer-like* Cayley codes and *High-locality* Cayley codes [1],[4],[14] (or equivalently, *Deletion codes* and *Transformation codes* [11]).

The Cayley codes in the first category — namely, the Prüfer Code [17] and the 'Prüfer-like' codes of Neville [12] and Deo & Micikevicius [5] — perform poorly as GA representations because they have low locality [6],[19],[20],[21].

The nine Cayley codes in the second category — namely, the Blob Code, Dandelion Code, and Happy Code described by Picciotto [16], the MHappy Code devised by Caminiti and Petreschi [2], and the other five 'Dandelion-like' codes given in [15] — have much higher locality than those in the first category, and exhibit superior performance when deployed as GA representations [2],[7],[11],[14],[15],[20],[21].

Although the Blob Code has lower locality than the eight Dandelion-like codes in this second category [15],[20],[21], research has shown that it is still an effective GA representation. For instance, in 2005, Julstrom showed that the Blob

Code is competitive with the 'edge-sets' representation on the minimum routing cost spanning tree problem [8].

In this paper, we develop a number of novel locality results concerning the Blob Code, building on previous work by Julstrom [7] and Thompson [20].

1.2 Essential notation and terminology

A *Cayley string* of order n is a string of $n-2$ integers from the set $[1,n]=\{1,2,\ldots,n\}$, with repeats allowed. For notational convenience, the $n-2$ elements in such a string will be indexed from 2 to $n-1$ rather than from 1 to $n-2$. The symbol $\mathcal{C}_n$ denotes the set of Cayley strings of order n.

The symbol $\mathcal{T}_n$ denotes the set of trees on the vertex set $[1,n]$ (or equivalently, the set of spanning trees of the complete graph K_n, where the vertex set is $[1,n]$).

It is well-known that $|\mathcal{T}_n| = n^{n-2}$ for each integer $n \geq 2$; this is *Cayley's formula* [3],[16]. Therefore, $|\mathcal{T}_n| = |\mathcal{C}_n|$.

A *Cayley code* is a bijective mapping between $\mathcal{T}_n$ and $\mathcal{C}_n$. Therefore, for each integer $n \geq 2$, there are $(n^{n-2})!$ possible Cayley codes [14]. When a Cayley code is deployed in a GA, the genotype space is $\mathcal{C}_n$ and the phenotype space is $\mathcal{T}_n$.

In this paper, we focus on a particular Cayley code known as the Blob Code (that is, we consider just one particular bijective mapping between the tree space $\mathcal{T}_n$ and the string space $\mathcal{C}_n$, as described in the next section). Under the Blob Code bijection, it is usual to refer to the strings in $\mathcal{C}_n$ as *Blob strings*, and we use this term in the rest of the paper.

Finally, we note that Picciotto [16] originally defined the Blob Code for trees on the vertex set $[0,n]$; in this paper, we redefine the Blob Code for trees on the vertex set $[1,n]$ by relabelling the vertices. Specifically, our formulation of the Blob Code is created by taking Picciotto's formulation on the vertex set $[0,n-1]$, and adding one to each vertex label; this procedure transforms the vertex set into $[1,n]$, with vertex 1 now playing the role of Picciotto's vertex 0.

2. THE BLOB CODE

In this section, we present decoding (string-to-tree) and encoding (tree-to-string) algorithms for the Blob Code, and illustrate these algorithms through an example.

The two algorithms described in this section both run in linear time, and are therefore quicker and easier than the traditional algorithms for the Blob Code, which require quadratic time in the worst case [16],[20].

The improved algorithms, first given in [15], are based on a little-known 1991 paper by Kreweras and Moszkowski [9], in which the authors present a tree code that is identical to the Blob Code — eight years before the work of Picciotto appeared. Recently, several other researchers [2],[11] have independently rediscovered the approach described in [9].

Later, we shall see that these alternative algorithms make it much easier to analyse the Blob Code's properties.

2.1 Decoding algorithm for the Blob Code

The algorithm below decodes any given Blob string in $\mathcal{C}_n$ into the corresponding tree in $\mathcal{T}_n$. As shown in [15], this algorithm can easily be implemented in linear time.

Input: A Blob string $B = (b_2, b_3, \ldots, b_{n-1}) \in \mathcal{C}_n$
Output: The tree $T \in \mathcal{T}_n$ corresponding to B

Step 1: Let G be the digraph whose vertex set is $[1,n]$ and whose edge set is $\{(i \to b_i) : i \in [2,n-1]\}$. Note that the edge set of G contains exactly $n-2$ directed edges.
Step 2: Colour each vertex $v \in [1,n]$ either black or white according to the following rule: vertex v is coloured black if none of the descendants of v in the digraph G has a label exceeding v, and coloured white otherwise. (The vertex w is a descendant of vertex v in the digraph G if and only if there is a directed path in G from v to w.)
Step 3: Label the black vertices as $x_1 < x_2 < \ldots < x_t$, where $t \in [2,n]$ is the total number of black vertices; observe that $x_1 = 1$ and $x_t = n$, since vertices 1 and n must be black.
Step 4: To construct the tree $T \in \mathcal{T}_n$ corresponding to B, take a set of n isolated vertices (labelled with the integers from 1 to n), create the edge (i, b_i) for each white vertex $i \in [2, n-1]$, create the edge $(x_i, b_{x_{i-1}})$ for each $i \in [3,t]$, and finally create the edge $(x_2, 1)$.

2.2 Encoding algorithm for the Blob Code

The algorithm presented in this subsection encodes any given tree $T \in \mathcal{T}_n$ as the corresponding Blob string $B \in \mathcal{C}_n$. Like the decoding algorithm in the previous subsection, this encoding algorithm can be implemented in linear time [15].

Input: A tree $T \in \mathcal{T}_n$
Output: The Blob string $B = (b_2, b_3, \ldots, b_{n-1}) \in \mathcal{C}_n$ that corresponds to B

Step 1: Treat the tree T as being rooted at vertex 1, and direct every edge towards this root to form the directed tree T'. For each $i \in [2,n]$, define $succ(i)$ such that $(i \to succ(i))$ is the unique directed edge leaving vertex i in T'.
Step 2: Colour each vertex $v \in [1,n]$ either black or white according to the following rule: vertex v is coloured black if none of the descendants of v in the directed tree T' has a label exceeding v, and coloured white otherwise. (The vertex w is a descendant of vertex v in the directed tree T' if and only if there is a directed path in T' from v to w.)
Step 3: Label the black vertices as $x_1 < x_2 < \ldots < x_t$, where $t \in [2,n]$ is the total number of black vertices; observe that $x_1 = 1$ and $x_t = n$, since vertices 1 and n must be black.
Step 4: To construct the Blob string $B \in \mathcal{C}_n$ corresponding to T, set $b_i = succ(i)$ for every white vertex $i \in [2,n-1]$, and set $b_{x_i} = succ(x_{i+1})$ for each $i \in [2, t-1]$.

2.3 An example of decoding and encoding

In this subsection we provide an example to illustrate the decoding and encoding algorithms described above.

Suppose we wish to decode the Blob string $B = (17,5,7,3,13,1,8,1,20,4,6,18,4,17,7,13,12,10) \in \mathcal{C}_{20}$ into the corresponding tree $T \in \mathcal{T}_{20}$. Following Step 1 of the decoding algorithm, we form the directed graph G on the vertex set $[1,20]$ that has 18 directed edges: $(2 \to 17)$, $(3 \to 5)$, $(4 \to 7), \ldots, (18 \to 12)$, $(19 \to 10)$. In Step 2, we find that there are ten black vertices (namely, 1, 5, 7, 8, 9, 11, 14, 16, 18, 20), and Step 3 tells us to assign the labels x_1 through to x_{10} to these vertices. The other ten vertices (namely, 2, 3, 4, 6, 10, 12, 13, 15, 17, 19) are white. Finally, in Step 4, we form the tree T corresponding to B. The white vertices provide the ten edges $(2,17)$, $(3,5)$, $(4,7)$, $(6,13)$, $(10,20)$, $(12,6)$, $(13,18)$, $(15,17)$, $(17,13)$, and $(19,10)$; the black vertices provide the eight edges $(7,3)$, $(8,1)$, $(9,8)$, $(11,1)$, $(14,4)$, $(16,4)$, $(18,7)$, and $(20,12)$, along with the additional edge $(5,1)$. Together, these nineteen edges constitute the tree $T \in \mathcal{T}_{20}$ corresponding to the given Blob string $B \in \mathcal{C}_{20}$; this tree is shown in Figure 1 at the top of the following page.

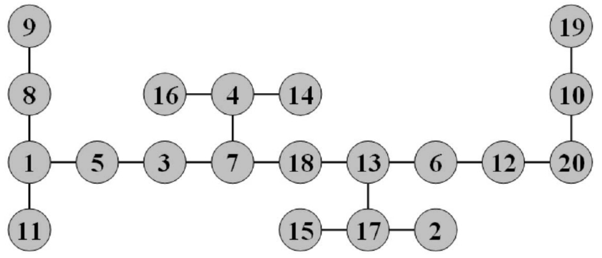


Figure 1: The tree $T \in \mathcal{T}_{20}$ that corresponds to the Blob string $B = (17, 5, 7, 3, 13, 1, 8, 1, 20, 4, 6, 18, 4, 17, 7, 13, 12, 10) \in \mathcal{C}_{20}$.

The Blob Code's encoding algorithm simply reverses the effect of the decoding algorithm. To illustrate this, we will reverse the example in the previous paragraph, by encoding the tree $T \in \mathcal{T}_{20}$ in Figure 1 as the corresponding Blob string $B \in \mathcal{C}_{20}$. Following Step 1 of the encoding algorithm, we direct all edges of T towards the vertex 1; this immediately shows us that $succ(2) = 17$, $succ(3) = 5$, $succ(4) = 7$, ..., $succ(19) = 10$, $succ(20) = 12$. Then, in Step 2, we easily recover the list of black vertices (namely, 1, 5, 7, 8, 9, 11, 14, 16, 18, 20), and in Step 3, we label these ten black vertices as $x_1, x_2, \ldots, x_{10}$. Clearly, the remaining ten vertices (namely, 2, 3, 4, 6, 10, 12, 13, 15, 17, 19) are white. Finally, in Step 4, we build the Blob string B. The white vertices provide ten elements of the string: $b_2 = 17$, $b_3 = 5$, $b_4 = 7$, $b_6 = 13$, $b_{10} = 20$, $b_{12} = 6$, $b_{13} = 18$, $b_{15} = 17$, $b_{17} = 13$, and $b_{19} = 10$. The black vertices provide the other eight elements: $b_5 = 3$, $b_7 = 1$, $b_8 = 8$, $b_9 = 1$, $b_{11} = 4$, $b_{14} = 4$, $b_{16} = 7$, and $b_{18} = 12$. Thus, the encoding algorithm recovers the Blob string $B = (17, 5, 7, 3, 13, 1, 8, 1, 20, 4, 6, 18, 4, 17, 7, 13, 12, 10) \in \mathcal{C}_{20}$ from the tree $T \in \mathcal{T}_{20}$ in Figure 1.

3. LOCALITY

3.1 The importance of locality

It is well-known that an effective GA representation must possess high locality — in other words, making small changes to the genotype should always lead to small changes in the corresponding phenotype. When a representation with low locality is used, similar strings in the genotype space may represent wildly different structures in the phenotype space, and the process of evolutionary search may hold negligible advantage over random search.

3.2 Distance metrics

Under the Blob Code, the genotype space (i.e., the space of Blob strings, $\mathcal{C}_n$) and the phenotype space (i.e., the space of trees, $\mathcal{T}_n$) both have natural notions of distance — and thus, natural notions of adjacency.

In the space of Blob strings, the simplest definition of the distance between two strings is the number of positions in which the strings differ (or equivalently, the number of single-element mutations required to transform one string into the other). Under this metric, the distance between two distinct Blob strings in $\mathcal{C}_n$ is always an integer in the range $[1, n-2]$, as each string in $\mathcal{C}_n$ has $n-2$ elements. We say that two Blob strings are 'adjacent' if the distance between them is one.

In the space of trees, the simplest definition of the distance between two trees T_1 and T_2 is the number of edges that belong to T_1 but not T_2 (or equivalently, the number of edge swaps required to transform one tree into the other); this distance metric is commonly known as the 'tree distance'. Under this metric, the distance between two distinct trees in $\mathcal{T}_n$ is always an integer in the range $[1, n-1]$, as each tree in $\mathcal{T}_n$ has $n-1$ edges. We say that two trees are 'adjacent' if the distance between them is one.

3.3 Mutation innovation

The locality of the Blob Code can be quantified using the concept of 'mutation innovation' [18], which describes the extent to which new phenotypic features arise when a given mutation operator is applied.

In general terms, the mutation innovation is the random variable Δ defined by the formula $\Delta = d_P(x, x^\star)$; that is, the phenotypic distance between a random solution x and a new solution $x^\star$ obtained by randomly mutating x.

In the context of the Blob Code, this mutation innovation formula may be rewritten as $\Delta = d_{\mathcal{T}}(T[B], T[B^\star])$, where:

- $d_{\mathcal{T}}()$ denotes the tree distance metric described in the previous subsection;
- B denotes a random Blob string;
- $B^\star$ denotes a Blob string obtained by making a random single-element mutation to the Blob string B;
- $T[B]$ and $T[B^\star]$ denote the trees corresponding to B and $B^\star$ respectively under the Blob Code.

Thus, Δ is simply the random variable which represents the number of tree edges that change when a random Blob string undergoes a random single-element mutation. (Single-element mutation is the most natural mutation operator for us to consider, since the distance metric in the genotype space is defined in terms of this operator.)

Observe that the distribution of Δ depends on n — in particular, Δ only takes values in the range $[1, n-1]$. Since high locality is advantageous, small values of Δ are desirable.

We define a mutation to be 'perfect' or 'optimal' if its associated value of Δ is one (i.e., if the mutation causes just a single edge-change in the corresponding tree). Perfect mutations are highly desirable, as they mean that a minimal step in the genotype space $\mathcal{C}_n$ causes a minimal step in the phenotype space $\mathcal{T}_n$.

Conversely, a mutation is 'pessimal' if its associated value of Δ is $n-1$ (i.e., if the mutation changes every edge of the corresponding tree). Pessimal mutations are highly undesirable, as they mean that a minimal step in the genotype space $\mathcal{C}_n$ causes a maximal step in the phenotype space $\mathcal{T}_n$.

Previous research [7],[20] into the locality of the Blob Code has considered only simple measures, such as the expected mutation innovation $E(\Delta)$, or the maximum value of Δ that is observed over a large number of mutations. In the next section, we significantly extend this work by investigating the distribution of Δ for different values of n.

4. LOCALITY OF THE BLOB CODE

In this section, we present exact and empirical locality measurements for the Blob Code, and then discuss the key features of the resulting data.

Table 1: The exact frequency distribution of Δ for small n.

n	$\Delta = 1$	$\Delta = 2$	$\Delta = 3$	$\Delta = 4$	$\Delta = 5$	$\Delta = 6$	$\Delta = 7$	$\Delta = 8$	$\Delta = 9$	*Total*
3	6	0								6
4	80	16	0							96
5	1116	356	26	2						1500
6	17974	6862	930	144	10					25920
7	334494	137602	25668	5632	768	46				504210
8	7100806	2997954	685364	184996	36198	4476	254			11010048
9	169650116	71684686	18928836	5847936	1441096	261934	30034	1626		267846264
10	4508332630	1880489456	553908190	187865610	54400364	12632554	2126892	232594	11710	7200000000

Table 2: The proportional distribution of Δ for small n. All figures are given to 5 d.p.

n	$E(\Delta)$	$\Delta = 1$	$\Delta = 2$	$\Delta = 3$	$\Delta = 4$	$\Delta = 5$	$\Delta = 6$	$\Delta = 7$	$\Delta = 8$	$\Delta = 9$
3	1.00000	100.00000	0.00000							
4	1.16667	83.33333	16.66667	0.00000						
5	1.27600	74.40000	23.73333	1.73333	0.13333					
6	1.35471	69.34414	26.47377	3.58796	0.55556	0.03858				
7	1.41478	66.34022	27.29061	5.09074	1.11699	0.15232	0.00912			
8	1.46252	64.49387	27.22925	6.22490	1.68025	0.32877	0.04065	0.00231		
9	1.50160	63.33862	26.76337	7.06705	2.18332	0.53803	0.09779	0.01121	0.00061	
10	1.53433	62.61573	26.11791	7.69317	2.60924	0.75556	0.17545	0.02954	0.00323	0.00016

4.1 Exact enumeration results for small *n*

For any given $n \geq 3$, there are exactly $n^{n-2}(n-1)(n-2)$ mutation events associated with the Blob Code — that is, n^{n-2} possible choices for the original Blob string $B \in \mathcal{C}_n$, $(n-2)$ possible choices for the position of B to experience mutation, and $(n-1)$ possible choices for the new value in that position.

Each of these $n^{n-2}(n-1)(n-2)$ mutation events has an associated value of Δ (i.e., the tree distance between the tree corresponding to the original string and the tree corresponding to the mutated string). For each $n \in [3, 10]$, these $n^{n-2}(n-1)(n-2)$ cases were exhaustively examined, and the distribution of Δ was constructed.

Table 1 shows the exact frequency distribution of Δ for each $n \in [3, 10]$; we refer to this as the 'locality signature' of the Blob Code.

To facilitate comparison between the data obtained for each n, Table 2 shows the expected value of Δ for each $n \in [3, 10]$ (rounded to 5 decimal places), along with the 'proportional' distribution of Δ for each $n \in [3, 10]$ (with all figures expressed as percentages to 5 decimal places).

4.2 Empirical results for large *n*

When n is greater than 10, exact enumeration becomes too computationally expensive, and the distribution of Δ must be estimated empirically through examining a large number of random mutation events.

In our experiments, we examined 12 larger values of n, chosen to lie roughly on a logarithmic scale: 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, and 100000. (Since previous studies have not considered values of n larger than 10000, we believed it would be useful to gain an insight into the asymptotic behaviour of Δ through examining some extreme values of n.)

For each of these 12 values of n, a large number of independent runs were performed (10^7 runs for each $n \leq 2000$, and 10^6 runs for each $n > 2000$). In each run, a Blob string $B \in \mathcal{C}_n$ was generated uniformly at random, and a random single-element mutation was applied to B to form a new Blob string $B^\star$. (Note that the term 'random single-element mutation' means a single-element mutation generated uniformly at random from the set of all $(n-1)(n-2)$ possible single-element mutations.) The trees $T \in \mathcal{T}_n$ and $T^\star \in \mathcal{T}_n$ corresponding to B and $B^\star$ under the Blob Code were then determined, and the associated value of Δ recorded. Therefore, for each value of n under study, we generated a large number (10^6 or 10^7) of independent realizations of the mutation innovation Δ.

To ensure the reliability of our simulation results, all of our experiments used pseudo-random numbers produced by a fifth-order multiple recursive generator with a period of around 10^{46}, as described by L'Ecuyer *et al.* [10]

Table 3 summarises the key data obtained through these experiments. For each value of n investigated, the table shows the expected mutation innovation $E(\Delta)$; the value of $P(\Delta = i)$ for $i \in [1, 5]$ and the remaining probability $P(\Delta > 5)$, all expressed as percentages to 3 decimal places; the extreme percentiles of the distribution, which indicate the shape of the distribution's tail; and the maximum value of Δ observed in the sample. For ease of comparison, the table also shows the measurements associated with $n = 10$, calculated directly from the exact enumeration results in Tables 1 and 2.

4.3 Discussion of locality results

In this subsection, we discuss the key features of the exact and empirical locality results presented in Tables 1 to 3.

4.3.1 Perfect mutation probability

The exact enumeration results shown in Tables 1 and 2 prove that $P(\Delta = 1)$ decreases as n increases in the range $[3, 10]$. However, the figures presented in Table 3 indicate that $P(\Delta = 1)$ increases towards a value of one as n increases from 20 through to 100000.

Our empirical results lead us to conjecture that $P(\Delta = 1)$ tends to one (albeit rather slowly) as n tends to infinity; thus, in the terminology of [14], it appears that the Blob Code has 'asymptotically optimal locality'. Unfortunately, there does not seem to be a simple proof of this result, along the lines of the proof presented in [14] for the Dandelion Code, because the Blob Code mapping is much more unwieldy than that of the Dandelion Code.

Table 3: The proportional distribution of Δ for various values of n. All figures have been rounded to three decimal places. The final seven columns show the extreme tail percentiles for each distribution. The figures are based on 10^7 runs for each $n \leq 2000$, and 10^6 runs for each $n > 2000$; the dividing line appears in the table.

n	$E(\Delta)$	$\Delta = 1$	$\Delta = 2$	$\Delta = 3$	$\Delta = 4$	$\Delta = 5$	$\Delta > 5$	95%	97.5%	99%	99.5%	99.75%	99.9%	*Max*
10	1.534	62.616	26.118	7.693	2.609	0.756	0.208	3	4	4	5	5	6	9
20	1.706	62.409	19.908	9.501	4.207	2.164	1.812	4	5	6	7	8	9	15
50	1.843	67.127	12.676	9.001	3.820	2.783	4.592	5	7	9	10	12	14	26
100	1.908	71.321	8.985	8.160	2.957	2.673	5.905	6	8	11	14	16	18	39
200	1.945	75.391	6.404	7.315	2.130	2.415	6.347	7	10	14	17	20	24	57
500	1.973	80.072	4.145	6.332	1.296	2.045	6.110	7	11	18	23	28	35	83
1000	1.985	83.092	2.988	5.681	0.855	1.787	5.597	7	12	21	28	36	46	130
2000	1.991	85.638	2.169	5.087	0.562	1.567	4.976	5	12	24	34	45	58	182
5000	1.992	88.486	1.416	4.345	0.313	1.313	4.128	5	11	26	42	58	80	208
10000	1.987	90.273	1.055	3.819	0.193	1.130	3.529	3	9	27	47	70	100	329
20000	2.014	91.741	0.739	3.360	0.124	0.983	3.053	3	7	26	52	84	128	469
50000	1.996	93.327	0.496	2.841	0.067	0.811	2.458	3	5	22	51	93	162	647
100000	1.991	94.444	0.341	2.427	0.043	0.691	2.054	3	5	17	47	101	191	1050

A detailed analysis of the perfect mutation probabilities in Table 3, using a log-log plot, suggests that $P(\Delta = 1)$ follows the relation $P(\Delta = 1) \approx 1 - n^{-0.25}$ for large n. For the three largest values of n (namely, 20000, 50000, and 100000), this formula evaluates to 0.91591, 0.93313, and 0.94377, each of which is within 0.2% of the corresponding empirical value shown in Table 3.

When the results in Table 3 are compared to those in [15], it is apparent that $P(\Delta = 1)$ approaches one much more slowly for the Blob Code than it does for any of the eight Dandelion-like codes (including the Dandelion Code, Happy Code, and MHappy Code). Indeed, for each of these eight codes, $P(\Delta = 1)$ is around 0.85 when $n = 100$, around 0.95 when $n = 1000$, and around 0.98 when $n = 10000$ [15].

Finally, additional numerical experiments indicate that the global minimum of $P(\Delta = 1)$ is 0.61759 (correct to 5 decimal places) at the point $n = 14$, and it appears that $P(\Delta = 1)$ increases monotonically as one moves away from this point in either direction. Thus, for any value of n, the Blob Code provides us with a perfect mutation probability exceeding 61.7%. The corresponding figure for the Dandelion Code is marginally higher — approximately 69.9% [14].

4.3.2 *Existence of pessimal mutations*

The exact enumeration results in Tables 1 and 2 show that, under the Blob Code, pessimal mutations cannot arise when $n = 3$ or $n = 4$, but do arise for each value of n in the range $[5, 10]$. Based on this observation, it is natural to wonder whether pessimal mutations can arise for larger n. The following theorem shows that the answer is 'yes'.

THEOREM 1. *Pessimal mutations can arise for any value of n greater than four.*

The proof of this theorem, which is given in Appendix A.1., is made significantly easier by the improved linear-time decoding algorithm presented earlier.

Theorem 1 shows that, for any $n > 4$, the tail of the Δ distribution extends all the way to $\Delta = n - 1$ (although the probability mass present in this tail may be exceedingly small, as indicated by the maximum observed values shown in Table 3). Thus, the theoretical worst-case behaviour of the Blob Code is inferior to that of the Dandelion-like codes, for which the range of Δ is always $[1, 5]$, no matter what value of n we consider [14],[15].

4.3.3 *Expected mutation innovation*

The exact enumeration results in Tables 1 and 2 prove that the expected mutation innovation $E(\Delta)$ increases as n increases in the range $[3, 10]$, and Table 3 suggests that $E(\Delta)$ continues to increase steadily as n increases further, stabilising at a value of around 2.000 for large n (allowing for fluctuations caused by simulation error).

On the basis of these figures, we conjecture that $E(\Delta)$ increases monotonically with n, and that the theoretical asymptoptic value of $E(\Delta)$ is exactly equal to two.

The observation that $E(\Delta)$ stabilises for very large n is not evident from the results presented by Julstrom [7] and Thompson [20], due to the higher level of random error in their simulations. Indeed, these previous experiments relied on just 10^5 or even 10^4 runs, whereas our simulations are based on 10^6 or 10^7 runs (depending on the value of n).

Our results indicate that the locality behaviour of the Blob Code is significantly weaker than that of the eight Dandelion-like codes, in the sense that the Blob Code does not possess the property of 'asymptotically optimal expected locality' (i.e., $E(\Delta)$ does not tend to one as n tends to infinity) [14]. Indeed, the results in [15] show that, for any of the eight Dandelion-like codes, $E(\Delta)$ is around 1.27 when $n = 100$, around 1.11 when $n = 1000$, and around 1.04 when $n = 10000$; these measurements are significantly more favourable than the corresponding figures for the Blob Code.

4.3.4 *Alternating phenomenon*

The results in Table 3 also indicate that an alternating phenomenon manifests itself for large n; for instance, when $n = 100000$, it is apparent that $P(\Delta = 2) < P(\Delta = 1)$, $P(\Delta = 3) > P(\Delta = 2)$, $P(\Delta = 4) < P(\Delta = 3)$, and so on. In particular, for large n (but not for small n), odd values of Δ are much more likely than even values of Δ.

Further analysis of the figures suggests that within each alternating distribution, two regular decay patterns manifest themselves — one for the even values of Δ, and one for the odd values of Δ (starting at $\Delta = 3$). However, we have not yet determined the underlying cause of this phenomenon.

4.4 Further analysis

4.4.1 *Nature of the distribution*

Under the Blob Code, we have observed the probability of perfect mutation $P(\Delta = 1)$ approaches one for large n. If the range of Δ were bounded above by a constant (independent of n), then the convergence of $P(\Delta = 1)$ to one would imply that $E(\Delta)$ also converges to one (that is, asymptotically optimal locality and bounded locality together guarantee asymptotically optimal expected locality.)

However, for the Blob Code, such a bound on Δ does not exist (since pessimal mutations exist for each $n > 4$), and there is therefore no guarantee that $E(\Delta)$ approaches one.

In fact, the figures in Table 3 show that the value of $E(\Delta)$ approaches *two* for large n; this is because the probability that Δ is large is not negligible, and compensates for the increasing probability mass at $\Delta = 1$.

4.4.2 Significance of 'black & white' configuration

When a Blob string undergoes mutation, the number of edge-changes caused by the mutation is intimately linked to changes in the black and white configuration of the vertices.

Suppose we define a mutation to be 'colour-preserving' if each vertex $v \in [1, n]$ has the same colour before the mutation as after the mutation (where a vertex's colour is either black or white). It is then easy to see that any colour-preserving mutation must also be a perfect mutation.

Although a mutation may still be perfect even if it is not colour-preserving, additional numerical simulations indicate that the proportion of perfect mutations that are *not* colour-preserving is very small (around $1/n$).

Thus, it appears that the probability of perfect mutation $P(\Delta = 1)$ approaches one as n increases because the proportion of mutations that are colour-preserving approaches one as n increases.

5. EFFECT OF MUTATION POSITION

So far, we have considered the distribution of Δ over the set of all $n^{n-2}(n-1)(n-2)$ possible mutation events. In this section, we show that the distribution of Δ strongly depends on the choice of mutation position $\mu \in [2, n-1]$.

5.1 A combinatorial result

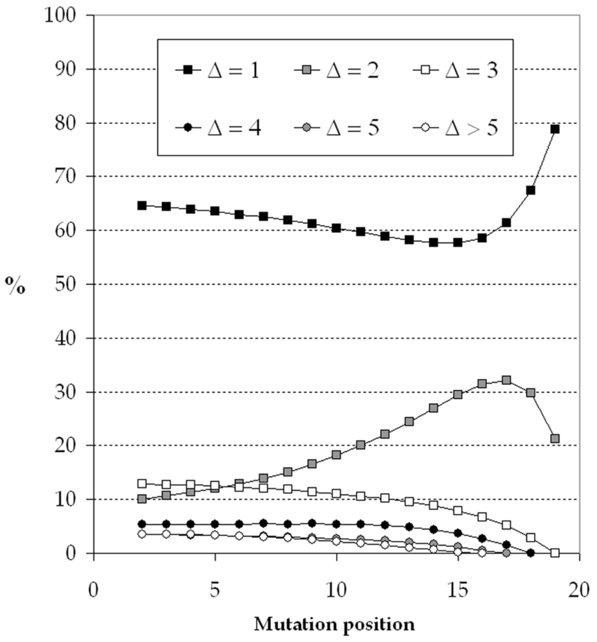
We begin the section with a theorem which indicates that fixing the mutation position $\mu \in [2, n-1]$ limits the possible values of Δ to the range $[1, n+1-\mu]$. This neat relationship was conjectured — but not proved — by Thompson [20]. For convenience, the theorem's proof is given in Appendix A.2.

THEOREM 2. *If we perform the single-element mutation $b_\mu \rightsquigarrow b_\mu^\star$ on the Blob string $B = (b_2, b_3, \ldots, b_{n-1}) \in \mathcal{C}_n$ to create the mutated Blob string $B^\star$ (where $\mu \in [2, n-1]$ and $b_\mu^\star \neq b_\mu$), then the tree $T \in \mathcal{T}_n$ corresponding to B and the tree $T^\star \in \mathcal{T}_n$ corresponding to $B^\star$ differ in at most $n+1-\mu$ edges (that is, T and $T^\star$ have at least $\mu - 2$ common edges).*

This theorem demonstrates that whenever a Blob string is mutated in position $\mu \in [2, n-1]$, the resulting number of edge-changes (i.e., the value of Δ) always lies in the range $[1, n+1-\mu]$. In particular, mutating the rightmost element of a Blob string can never change more than two edges in the corresponding tree; conversely, every pessimal mutation must involve changing the leftmost element of a Blob string.

5.2 Some empirical results

We now report additional experimental results that give further insight into the effect of mutation position on the distribution of Δ.



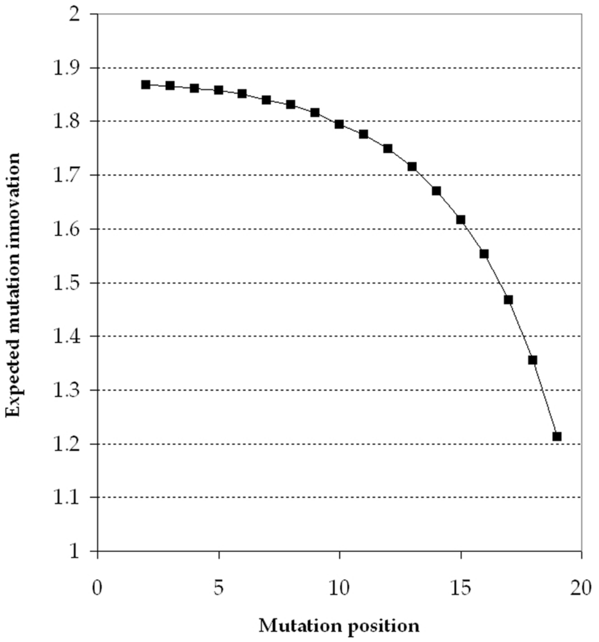


Figure 2: The upper graph shows how the distribution of Δ varies with mutation position μ. The lower graph shows how the expected mutation innovation $E(\Delta)$ decreases as the mutation position μ increases. In both cases, $n = 20$, and so $\mu \in [2, 19]$.

For the purposes of these experiments, we focused on the case $n = 20$. For each possible mutation position $\mu \in [2, 19]$, we performed 10^6 independent runs. In each run, a Blob string $B \in \mathcal{C}_{20}$ was generated uniformly at random, and a random mutation was applied to element b_μ to form a new Blob string $B^\star$ (with the new value $b^\star_\mu$ being chosen uniformly at random from the nineteen elements in the set $[1, 20]\backslash\{b_\mu\}$). The trees $T \in \mathcal{T}_{20}$ and $T^\star \in \mathcal{T}_{20}$ corresponding to B and $B^\star$ under the Blob Code were then determined, and the associated value of Δ recorded. Therefore, for each $\mu \in [2, 19]$, we generated 10^6 independent realizations of Δ.

The upper graph in Figure 2 shows the distribution of Δ observed for each mutation position $\mu \in [2, 19]$; the lower graph in Figure 2 shows the expected mutation innovation $E(\Delta)$ for each value of $\mu \in [2, 19]$.

The two graphs have a number of noteworthy features. In the upper graph, we see that the probability of perfect mutation $P(\Delta = 1)$ takes its minimum value (around 57.6%) at the point $\mu = 14$, and rises as one moves away from this point in either direction, taking its maximum value (around 78.8%) when $\mu = 19$. Conversely, $P(\Delta = 2)$ takes its minimum value (around 10.0%) when $\mu = 2$, rises to its maximum value (around 32.1%) at the point $\mu = 17$, and then falls again as μ increases further. The other four curves decrease steadily towards zero as μ increases in the range $[2, 19]$. Note that, in accordance with Theorem 2, the curves for $P(\Delta = 3)$, $P(\Delta = 4)$, $P(\Delta = 5)$, and $P(\Delta > 5)$ reach zero at $\mu = 19$, $\mu = 18$, $\mu = 17$, and $\mu = 16$ respectively (and remain at zero for all larger values of μ). In the lower graph, we see that the expected mutation innovation $E(\Delta)$ decreases from around 1.87 to around 1.21 as the value of μ increases in the range $[2, 19]$, with the rate of decrease becoming steadily greater as μ increases.

Further experiments revealed that the qualitative features of all these curves are similar for other values of n; however, two additional trends should be briefly noted. Firstly, as n increases, the minimum point of $P(\Delta = 1)$ and the maximum point of $P(\Delta = 2)$ shift further to the right in relative terms — for instance, when $n = 1000$, the minimum of $P(\Delta) = 1$ occurs when $\mu \approx 970$ (i.e., 97% of the way along the Blob string) and the maximum of $P(\Delta) = 2$ occurs when $\mu \approx 980$ (i.e., 98% of the way along the Blob string). Secondly, for much larger values of n, it was noted that the curve corresponding to $P(\Delta = 4)$ exhibits a rise-and-fall pattern similar to that of $P(\Delta = 2)$, but very much smaller in magnitude; further tests may show that this phenomenon extends to other even values of Δ.

5.3 Discussion

Theorem 2 establishes that the choice of the mutation position $\mu \in [2, n - 1]$ places restrictions on the range of Δ — specifically, the maximum possible value of Δ decreases as μ increases. The empirical results in the previous subsection build on this result by showing how the distribution of Δ changes with μ, using the case $n = 20$ as an example.

Our results show that, on average, larger values of μ are associated with more desirable mutations, both in terms of the expected mutation innovation $E(\Delta)$ and the maximum value of Δ that can arise (as both of these measures decrease as μ increases). In terms of the perfect mutation probability $P(\Delta = 1)$, we find that the largest (and thus, most favourable) value occurs when $\mu = n - 1$, but larger values of μ do not necessarily give larger values of $P(\Delta = 1)$.

However, while all three of the above locality measures identify $\mu = n - 1$ as the most desirable mutation position, it would be foolish to conclude that we should perform mutations exclusively in this position — of course, our mutation operator should allow random variation to be introduced in all string positions, and allow any Blob string to be mutated into any other through a suitable series of mutations.

Nonetheless, our results do raise an interesting prospect: when working with the Blob Code (or indeed, any other Cayley code), it may be possible to enhance the locality of the standard single-element mutation operator (under which the mutation position is selected uniformly at random from $[2, n - 1]$) by introducing a positional bias to exploit the superior locality at different string positions. This idea is a topic of ongoing research.

6. CONCLUSION AND FUTURE WORK

In this paper, we have extended previous research into the locality properties of the Blob Code, by establishing a number of different exact and empirical results relating to the distribution of the mutation innovation Δ.

Our experiments showed that the Blob Code possesses the desirable property of asymptotically optimal locality (i.e., the perfect mutation probability $P(\Delta = 1)$ approaches one as n grows). However, we also found that pessimal mutations (i.e., mutations for which $\Delta = n - 1$) can arise under the Blob Code for any $n > 4$. Consequently, the Blob Code does not possess asymptotically optimal expected locality: as n becomes large, the expected mutation innovation $E(\Delta)$ approaches two, rather than the ideal value of one. These results confirm that the locality of the Blob Code is high, but not as high as that of the eight Dandelion-like codes.

We also studied the effect of the mutation position μ on the distribution of Δ, and found that mutation positions towards the end of the Blob string exhibited the highest locality; it might be possible to exploit this phenomenon.

Finally, in terms of future work, it would be valuable to perform a detailed comparison of the locality of the Blob Code and that of other Cayley codes, building on the work in this paper. This investigation could also consider alternative mutation operators, such as swap mutation. It would also be interesting to see how the mutation position μ affects the distribution of Δ for each of the different Cayley codes.

7. REFERENCES

[1] S. Caminiti, I. Finocchi, and R. Petreschi. A unified approach to coding labeled trees. In *Lecture Notes in Computer Science 2976 — Proceedings of LATIN 2004*, pages 339–348, Buenos Aires, April 2004.

[2] S. Caminiti and R. Petreschi. String coding of trees with locality and heritability. In *Lecture Notes in Computer Science 3595 — Proceedings of COCOON 2005*, pages 251–262, Kunming, August 2005.

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

[4] N. Deo and P. Micikevicius. Prüfer-like codes for labeled trees. *Congressus Numerantium*, 151:65–73, 2001.

[5] N. Deo and P. Micikevicius. A new encoding for labeled trees employing a stack and a queue. *Bulletin of the Institute of Combinatorics and its Applications*, 34:77–85, 2002.

[6] J. Gottlieb, B. A. Julstrom, G. R. Raidl, and F. Rothlauf. Prüfer numbers: A poor representation of spanning trees for evolutionary search. Technical Report 2001001, Illinois Genetic Algorithms Laboratory (IlliGAL), January 2001.

[7] B. A. Julstrom. The Blob Code: A better string coding of spanning trees for evolutionary search. In *GECCO Workshop Program*, pages 256–261, San Mateo, California, 2001.

[8] B. A. Julstrom. The Blob Code is competitive with edge-sets in genetic algorithms for the minimum routing cost spanning tree problem. In *Proceedings of GECCO 2005*, pages 585–590, Washington DC, 2005.

[9] G. Kreweras and P. Moszkowski. Tree codes that preserve increases and degree sequences. *Discrete Mathematics*, 87(3):291–296, 1991.

[10] P. L'Ecuyer, F. Blouin, and R. Coutre. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 3:87–98, 1993.

[11] P. Micikevicius, S. Caminiti, and N. Deo. Linear-time algorithms for encoding trees as sequences of node labels. To appear in Congressus Numerantium.

[12] E. H. Neville. The codifying of tree structure. In *Proceedings of the Cambridge Philosophical Society*, volume 49, pages 381–385, 1953.

[13] C. C. Palmer and A. Kershenbaum. Representing trees in genetic algorithms. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, volume 1, pages 379–384, June/July 1994.

[14] T. Paulden and D. K. Smith. From the Dandelion Code to the Rainbow Code: A class of bijective spanning tree representations with linear complexity and bounded locality. *IEEE Transactions on Evolutionary Computation*, 10(2):108–123, April 2006.

[15] T. Paulden and D. K. Smith. Recent advances in the study of the Dandelion Code, Happy Code, and Blob Code spanning tree representations. In *Proceedings of the 2006 IEEE World Congress on Computational Intelligence [DVD-ROM only]*, pages 7464–7471, Vancouver, July 2006.

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

[17] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv der Mathematik und Physik*, 27:742–744, 1918.

[18] G. R. Raidl and J. Gottlieb. Empirical analysis of locality, heritability and heuristic bias in evolutionary algorithms: A case study for the multidimensional knapsack problem. *Evolutionary Computation*, 13(4):441–475, 2005.

[19] F. Rothlauf. *Representations for Genetic and Evolutionary Algorithms.* Physica-Verlag, Heidelberg, Germany, 2002.

[20] E. Thompson. *The application of evolutionary algorithms to spanning tree problems.* PhD thesis, University of Exeter, United Kingdom, 2003.

[21] E. Thompson, T. Paulden, and D. K. Smith. The Dandelion Code: A new coding of spanning trees for genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 11(1):91–100, February 2007.

APPENDIX

A. PROOFS OF THEOREMS

In this section, we provide proofs of Theorems 1 and 2. In both cases, the proofs follow quickly from the improved linear-time decoding algorithm given earlier in the paper.

A.1 Theorem 1

Proof. We will prove that at least one pessimal mutation exists for each $n > 4$ by providing an explicit construction.

Given $n > 4$, consider the Blob string $B = (1, 2, \dots, n-2)$. The digraph G associated with B consists of the directed edges $(i \rightarrow i-1)$ for each $i \in [2, n-1]$. Thus, when the Blob Code decoding algorithm is applied, every vertex in G will be classified black. Thus, the tree T corresponding to B under the Blob Code consists of the edge $(i, i+2)$ for each $i \in [1, n-2]$, along with the single edge $(1, 2)$.

Now suppose that B is mutated in its leftmost position to produce the new string $B^\star = (n, 2, 3, \dots, n-2)$. It is clear that the mutated digraph $G^\star$ associated with $B^\star$ consists of the directed edges $(i \rightarrow i-1)$ for each $[3, n-1]$, along with the single directed edge $(2 \rightarrow n)$. Clearly, in the mutated digraph $G^\star$, each vertex in $[2, n-1]$ is white, and vertices 1 and n are black. It follows that the tree $T^\star$ corresponding to $B^\star$ under the Blob Code consists of the edges $(i, i+1)$ for i in $[2, n-2]$, along with the two edges $(1, n)$ and $(2, n)$.

Since T and $T^\star$ have no common edges, the mutation that transformed B into $B^\star$ is a pessimal mutation. Thus, at least one pessimal mutation exists for each $n > 4$. $\square$

A.2 Theorem 2

Proof. For any Blob string $B \in \mathcal{C}_n$, we wish to prove that mutating the value of b_μ causes at most $n+1-\mu$ edge-changes in the corresponding tree.

Firstly, we observe that any $i < \mu$ which was classified black prior to the mutation still remains black after the mutation. This is simple to prove: prior to the mutation, μ cannot have been a descendant of i in G (because $i < \mu$, and we know that i is black); thus, the descendants of i in G are identical to the descendants of i in $G^\star$, and so i must remain its original colour after the mutation — that is, black.

We then assert that any $i < \mu$ which was classified white prior to the mutation remains white after the mutation. To demonstrate this, we consider two mutually exclusive and exhaustive cases: (1) If μ was a descendant of i in G prior to the mutation, then μ remains a descendant of i in $G^\star$ after the mutation, and thus i remains white because $i < \mu$; (2) If μ was not a descendant of i in G prior to the mutation, then the descendants of i in G are identical to the descendants of i in $G^\star$, and so i must remain its original colour after the mutation — that is, white.

We have therefore shown that the black or white status of each vertex $i < \mu$ is not altered by the mutation $b_\mu \rightsquigarrow b^\star_\mu$. It is then easy to see that during the decoding of $B^\star$, each $i < \mu$ will be joined to exactly the same vertex as during the decoding of B (namely, vertex b_i if i is white, or vertex $b_{x_{j-1}}$ if $i = x_j$ for some $j > 2$, or vertex 1 if $i = x_2$). Thus, there are guaranteed to be $\mu-2$ edges common to both the original tree T and the mutated tree $T^\star$ — in other words, mutating the value of b_μ can cause at most $n+1-\mu$ edge-changes in the corresponding tree. $\square$