String coding of trees with locality and heritability

Saverio Caminiti and Rossella Petreschi

Dipartimento di Informatica, Università degli Studi di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy.
{caminiti, petreschi}@di.uniroma1.it

Abstract. We consider the problem of coding labelled trees by means of strings of vertex labels and we present a general scheme to define bijective codes based on the transformation of a tree into a functional digraph. Looking at the fields in which codes for labelled trees are utilized, we see that the properties of locality and heritability are required and that codes like the well known Prüfer code do not satisfy these properties. We present a general scheme for generating codes based on the construction of functional digraphs. We prove that using this scheme, locality and heritability are satisfied as a direct function of the similarity between the topology of the functional digraph and that of the original tree. Moreover, we also show that the efficiency of our method depends on the transformation of the tree into a functional digraph. Finally we show how it is possible to fit three known codes into our scheme, obtaining maximum efficiency and high locality and heritability.

1 Introduction

Labeled trees are of interest in both practical and theoretical areas of computer science. To take just two examples: Ethernet has a unique path between terminal devices, labeling the tree vertices is necessary to identify each device in the network without ambiguity; trees are used in biology to represent phylogenetic relationships between species, populations, individuals, or genes represented by labels.

Coding labeled trees by means of strings of vertex labels is an interesting alternative to the usual representations of tree data structures in computer memories, since it has many practical applications [3]. Evolutionary algorithms over trees maintain a population of data structures that represents candidate solutions to a problem. The association between structures and solutions is realized through a decoder which must exhibit efficiency, locality, and heritability if the evolutionary search is to be effective [4, 5, 7, 12]. In this context it is possible to show that representing a tree as a string increases the probability to guarantee the required properties will be attained.

Furthermore, string base coding makes it possible to generate random uniformly distributed trees and random connected graphs [9]. Indeed the generation of a random string, followed by the use of a fast decoding algorithm, is typically more efficient than generating a tree by adding edges randomly, where one must be careful not to introduce cycles.

Finally, tree codes are also used for data compression and in the computation of forest volumes of graphs [8].

Unless stated otherwise, here we will consider the tree as rooted in vertex 0 and its n vertices labeled from 0 to $n-1$.

The *naïve* method for relating a tree to a string P consists in associating to each vertex x the value of its parent $p(x)$; P has cardinality $n-1$ since the root node 0 can be omitted, in the following we refer to the naïve string as parent array. It should be noted that an arbitrary string of length $n-1$ over $[0, n-1]$ is not necessarily a tree, but it may be either a non-connected or a cyclic graph.

We are interested in those types of coding that define a bijection between the set of labeled trees of n vertices and a set of strings over $[0, n-1]$. Since Cayley has proved that the number of labeled trees on n vertices is n^{n-2}, we know that this kind of one-to-one correspondence requires the cardinality of the string to be equal to $n-2$ [1].

In his proof of Cayley's theorem, Prüfer provided the first bijective string based coding for trees [11]. Over the years since then many codings behaving like that of Prüfer have been introduced. In [2] a complete survey on these codes is presented, and it is shown that coding and decoding in sequential linear time is possible using each of these codes; efficient parallel algorithms are also presented.

However, even if they are extremely efficient, Prüfer-like codes lack other desirable properties, such as locality and heritability, as noted in [6]. An experimental analysis [7] shows that these properties are much better satisfied by the Blob code, defined by Picciotto [10].

In this paper we present a general scheme for defining bijective codes based on the transformation of a tree into a functional digraph. We also show how the properties of locality and heritability are related to differences between the digraph and the original tree. Then we highlight the differences between Prüfer-like codes and codes derivable from our scheme. Finally, we show how it is possible to map some known codes, including the Blob code, to our scheme.

2 Preliminaries

In this section, we introduce some definitions that will be needed in the rest of the paper.

Definition 1. *Given a function g from the set $[0, n]$ to the set $[0, n]$, the* functional digraph $G = (V, E)$ *associated with g is a digraph with $V = \{0, \ldots, n\}$ and $E = \{(v, g(v))$ for every $v \in V\}$.*

For this class of graphs the following lemma holds:

Lemma 1. *A digraph $G = (V, E)$ is functional if and only if $|E| = n$ and the outer degree of each vertex is equal to 1.*

Corollary 1. *Each connected component of a functional digraph is composed of several trees, each of which is rooted in a vertex belonging to the* core *of the component, which is either a cycle or a loop (see Figure 1a).*

Functional digraphs are easily generalizable for the representation of functions which are undefined in some values: if $g(x)$ is not defined, the vertex x in G does not have any outgoing edge. The connected component of G containing an x, such that $g(x)$ is not defined, is a tree rooted in x without cycles or loops (see Figure 1b).

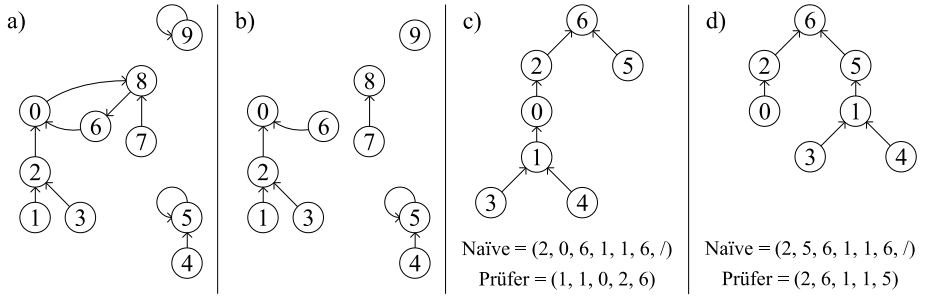

Fig. 1. a) A functional digraph associated with a fully defined function; b) A functional digraph associated with a function undefined in 0, 8, and 9; c) A tree T, and the corresponding naïve and Prüfer codes (notice that this tree is not rooted at 0 and then the naïve code has cardinality n); d) $T' = T - (1, 0) + (1, 5)$ and the corresponding naïve and Prüfer codes.

Definition 2. *A* labeled n-tree *is an unrooted tree with n vertices, each with a distinct label selected in the set* $[0, n-1]$.

Definition 3. *In a labeled n-tree, the set of vertices between a vertex v to a vertex u is called the* path *from v to u; u and v do not belong to the path.*

Since in this paper we deal only with labeled trees, we will refer to them simply as trees. In the following, when it is necessary to root a tree in one of its vertices, we will consider its edges oriented upwards from leaves to root.

Remark 1. Let T be a rooted tree and $p(v)$ be the parent of v for each v in T. T is the functional digraph associated with the function p.

Let us call *n-string* a string of n elements in the set $[0, n+1]$.

Definition 4. *A code is a method for associating trees to strings in such a way that different trees yield different strings. A* bijective code *is a code associating n-trees to $(n-2)$-strings.*

Below, when there is no risk of confusion, we will identify a tree with its associated string, and vice versa.

A code satisfies the *Locality Property* if small changes in the tree correspond to small changes in the associated string, and vice versa.

In evolutionary algorithms, where sometimes a new string is generated by mixing two existing strings, another desirable property is the *Heritability Property*: edges of the tree corresponding to the mixed string belong to one of the two existing trees.

Let us look at the naïve code representing a tree with the parent vector. Since each edge of a tree corresponds to an element of the string, this code exhibits maximal locality: a single change in the tree corresponds to a single change in the associated string, and vice versa (see Figure 1c and 1d). Naïve code also maximally satisfies heritability: in each string the i-th element corresponds to the edge $(i, p(i))$ of the tree, it implies that a tree obtained by mixing two existing strings has only edges coming from the two existing trees. Unfortunately, this code is not bijective, so a string obtained by modifying one or more strings is

not necessarily a tree: more precisely, the probability of obtaining a tree is $\frac{1}{n}$. This is a serious shortcoming of naïve code.

The *Prüfer* code proceeds recursively, deleting the leaf with smallest label from the tree; when a leaf is deleted, the label of its parent is added to the code. This code is bijective, but exhibits extremely poor locality [6] (see Figure 1c and 1d).

3 General method

In this section we present a general method for defining bijections between the set of labeled n-trees and $(n-2)$-strings. Our idea is to modify the naïve method so as to reduce the dimension of the string that it yields.

In order to build an $(n-2)$-string, we conjecture that the tree is rooted at a fixed vertex x, and that there exists another fixed vertex y having x as parent. Under these assumptions, in the parent array representation we may omit the information related to both x and y. It is easy to root a given unrooted tree at a fixed vertex x, however it is not so clear how to guarantee the existence of edge (x, y). A function φ manipulates the tree in order to ensure the existence of (x, y) and this is what characterizes each specific instance of our general method. The function φ has to transform T into a functional digraph G, with $n-1$ edges associated with a function g, such that $g(x)$ is undefined and $g(y) = x$. Below we see the coding scheme when φ, x, and y are known:

General Coding Scheme
Input: Input: an n-tree T
Output: Output: an $(n-2)$-string C
1. Root T in x
2. Construct $G = \varphi(T)$
3. **for** $v = 0$ **to** $n-1$ **do**
4. **if** $(v \neq x$ **and** $v \neq y)$ **then** add $g(v)$ to C

To guarantee the bijectivity of the coding obtained, the function φ must be invertible; only under this hypothesis is it possible to define the decoding scheme:

General Decoding Scheme
Input: Input: an $(n-2)$-string C
Output: Output: an n-tree T
1. Reconstruct the graph G starting from code C
2. Add the fixed edge (x, y)
3. Compute $T = \varphi^{-1}(G)$

In our method, the topology of graph G directly identifies the string C, since for each vertex from 0 to $n-1$ its outgoing edge is considered. The obtained C is similar to the naïve code of the tree to precisely the same extent as the tree topology is similar to the graph topology. Since the naïve code has naturally maximal locality and heritability, if we are interested in obtaining high locality and heritability codes we have to look for those φ functions that minimize the variations introduced into the tree. Consequently the efficiency of our coding and decoding schemes is strictly dependent on the computation of φ.

It should be noted that in all Prüfer-like codes the tree topology determines the elimination order of vertices, so a small change in the tree may cause a variation of this order and thus a big change in the string (see Figure 1c and 1d). This is the reason why Prüfer and Prüfer-like codes exhibit low locality and heritability [6].

In the following, we show that several codes introduced in the literature [10] can be mapped into our general scheme, and we provide optimal computation for their φ functions.

4 Blob code

The *Blob* code was introduced by Picciotto [10] in her Ph.D. Thesis. The algorithm used to obtain a string starting from a tree is:

Blob Coding Algorithm
Input: Input: an n-tree T
Output: Output: an $(n-2)$-string C
1. Initialize $blob = \{n\}$, $C = ()$
2. Root T in 0
3. **for** $v = n-1$ **to** 1 **do**
4. **if** $((path(v,0) \bigcap blob) \neq \emptyset)$ **then** $C[v-1] = p(v)$
5. delete $(v, p(v))$ and insert v in $blob$
6. **else** $C[v-1] = p(blob)$
7. delete $(blob, p(blob))$ and add $(blob, p(v))$
8. delete $(v, p(v))$ and insert v in $blob$

In this algorithm *blob* is a macro-vertex, i.e. it has a parent but it contains many other vertices. Each vertex included in *blob* maintains its own subtree, if any, but this subtree is not necessarily included in the *blob*.

We will call *stable* all vertices satisfying the test in line 4; their corresponding value in the code is their original parent.

Analyzing this algorithm we can see that the line 4 condition is not tied to the incremental construction of the *blob*, but it can be globally computed in the initialization phase as the Lemma 2 asserts:

Lemma 2. *Stable vertices are all vertices v such that $v < max(path(v,0))$.*

Proof. At step v of main cycle the set *blob* contains all the vertices from $v+1$ to n. Then the condition of line 4 holds if and only if at least a vertex greater than v occurs in $path(v,0)$.

Note that $path(v,0)$ is a set as stated in Definition 3.

Lemma 3. *For each unstable vertex v, $p(z)$ is the corresponding value in the code, where z is $min\{u | u > v$ and u unstable$\}$.*

Proof. In line 6 the current parent of *blob* defines the code value corresponding to an unstable vertex v and in line 7 the *blob* becomes child of $p(v)$. It implies that when line 6 is executed for vertex v, $p(blob)$ is equal to the parent of the smaller unstable vertex greater than v, i.e. $p(z)$.

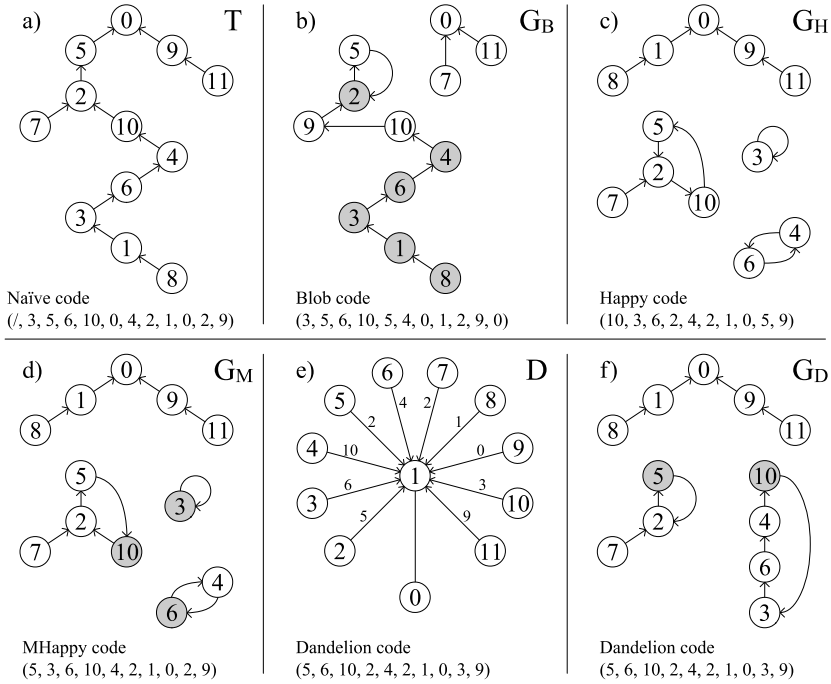

Fig. 2. a) A sample tree T rooted in 0; b) $G_B = \varphi_b(T)$, stable vertices are represented in gray; c) G_H computed from T by the original Happy Coding Algorithm; d) $G_M = \varphi_m(T)$, maximal vertices are represented in gray; e) D computed from T by the Dandelion Coding Algorithm; f) $G_D = \varphi_d(T)$, flying vertices are represented in gray.

Let us define a function φ_b constructing a graph G starting from a tree T in the following way: for each unstable vertex v, removes edge $(v, p(v))$ and add edge $(v, p(z))$ where $z = min\{u | u > v$ and u unstable$\}$. If z does not exist, i.e. when $v = n$, add the edge $(v, 0)$.

In Figure 2a and 2b a tree T and a graph $G = \varphi_b(T)$ are depicted.

Remark 2. Each path in T from a stable vertex v to $m = max(path(v, 0))$ is preserved in $G = \varphi_b(T)$.

Theorem 1. *It is possible to fit Blob code into our general scheme when $x = 0$, $y = n$, and $\varphi = \varphi_b$.*

Proof. It is trivial to see that graph $G = \varphi_b(T)$ is a functional digraph, since: a) each vertex has outdegree equal to 1; b) the function g associated with G is undefined in 0; c) $g(n) = 0$.

Lemmas 2 and 3 guarantee that the generated string C is equal to the code computed by Blob Coding Algorithm.

Now we have to prove that φ_b is invertible, i.e. we have to show how to rebuild T from G.

First we eliminate cycles from G, then we recompute stable and unstable vertices of original T to identify, according to Remark 2, those vertices that must recompute their parents in G.

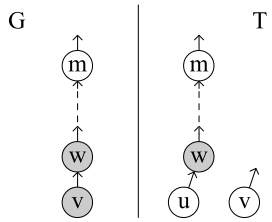

Fig. 3. Vertices involved in the proof of Theorem 1 both in G and in T. Stable vertices are represented in gray.

Each cycle Γ is broken deleting the edge outgoing from γ, the maximum label vertex in Γ. Remark 2 implies that γ was unstable in T, indeed if γ was stable in T the path from γ to $max(path(\gamma, 0))$ must appear in G, but this implies a vertex greater than γ in Γ. Notice that γ becomes the root of its own connected component, while 0 is the root of the only connected component not containing cycles. The identification of γ is a step towards the recomputation of stable and unstable vertices.

We call stable in G each vertex v such that $max(path(v, \gamma_v) \bigcup \{\gamma_v\}) > v$, where γ_v is the root of the connected component containing v.

The path preservation stated in Remark 2 guarantees that each vertex v, stable in T, is stable in G. Let us now prove that the vice versa is also true.
Let us assume, by contradiction, that there exists a vertex v stable in G but unstable in T. And let us call $m = max(path(v, \gamma_v) \bigcup \{\gamma_v\})$ in G. It holds $v < m$ and m unstable both in G and in T. In G m is unstable because there are not vertices greater than m in $path(v, \gamma_v) \bigcup \{\gamma_v\}$; in T m can not be stable because, as noted before, each stable vertex in T remains stable in G.

W.l.o.g. we assume that all vertices between v and m are stable both in G and in T. Let w be the parent of v in G. By definition of φ_b there exists a vertex $u > v$ unstable in T such that $p(u) = w$ in T. In Figure 3 v, m, u, and w are depicted both in G and in T.

Since m is in the path from u to 0 in T, m must be smaller than u. Then $v < m < u$ and m is unstable in T contradicting the assertion that there are no unstable vertices in T between v and u (by definition of φ_b).

The computational complexity of original Blob coding and decoding algorithms are quadratic, due to the computation of paths at each iteration. Our characterization of stable vertices (cfr. Lemma 2) decreases the complexity of coding algorithm to $O(n)$. Linear complexity for both coding and decoding can be obtained by fitting Blob code into our general scheme. Indeed both φ_b and φ_b^{-1} can be implemented in $O(n)$ sequential time: computation of the maximum vertex in the upper path (coding) and cycles identification (decoding) can both be implemented by simple search techniques.

In [7], an experimental analysis shows that locality and heritability are satisfied by the Blob code much better than by the Prüfer code. The reasons behind the experimental results become clear when Blob code is analyzed according to our method, which is quite different from Picciotto's original idea. The functional digraph generated by φ_b preserves an edge of the original tree for each stable vertex, and for these vertices $g(v) = p(v)$: this partial similarity with naïve code is the reason for the soundness of locality and heritability.

In the next two sections we discuss two codes that better exploit similarities with naïve code.

5 Happy code

Happy code was introduced in [10], and it appears with a structure completely different from the Blob code:

HAPPY CODING ALGORITHM
Input: Input: an n-tree T
Output: Output: an $(n-2)$-string C
1. Root T in 0 and initialize $J = p(1)$
2. **while** $p(1) \neq 0$ **do**
3. $j = p(1)$, delete $(1, j)$, delete $(j, p(j))$, and add $(1, p(j))$
4. **if** $j > J$ **then** $J = j$ and add (J, J)
5. **else** add $(j, p(J))$, delete $(J, p(J))$, and add (J, j)
6. **for** $v = 2$ **to** n **do** $C[v-2] = p(v)$

This algorithm focuses on the path from 1 to 0. Since the aim of the algorithm is to ensure the existence of edge $(1, 0)$, all the vertices on the original path from 1 to 0 are sequentially moved in order to form cycles. Let us call *maximal* each vertex v in $path(1, 0)$ such that $v > max(path(1, v))$. The first cycle is initialized with $p(1)$ and each time a maximal vertex is analyzed a new cycle is initialized (see Figure 2c).

Notice that the algorithm inserts a vertex j in a cycle immediately after J, the maximal vertex in the cycle. This implies that in the resulting graph the vertices in a cycle will be in reverse order with respect to their position in the original tree (see Figure 2c). Since we are interested in keeping the graph as close as possible to the original tree, we will consider a slightly modified version of this code which avoids this inversion: j is attached immediately before J instead of immediately after. Let us call this modified version of happy code *MHappy* code (see Figure 2d).

For MHappy code we define a function φ_m which, given a tree T, constructs a graph G in the following way: for each maximal vertex v in $path(1, 0)$ remove the edge incoming at v in this path, and add an edge (z, v) where z is the child of the next maximal vertex. If z does not exist, use the child of 0; finally remove the edge incoming at 0 in the path and add the edge $(1, 0)$.

Theorem 2. *It is possible to fit MHappy code into our general scheme when* $x = 0$*,* $y = 1$*, and* $\varphi = \varphi_m$*.*

Proof. It is trivial to see that the MHappy coding transforms T into the same functional digraph generated by φ_m: this corresponds to a function g undefined in 0 (the root) and is such that $g(1) = 0$ (edge $(1, 0)$).

To show that φ_m is invertible, first sort all cycles in G into increasing order with respect to their maximum vertex γ, then breack each cycle removing the edge incoming at γ. Since the order of cycles obtained is the same as that in which they where originally created, we rebuild the original tree inserting all the vertices of each cycle in the path from 1 to 0 in accordance with the order of the cycles.

φ_m and φ_m^{-1} can be implemented in $O(n)$ sequential time because coding requires the computation of maximal vertices in the path from 1 to 0 and decoding requires cycle identification and integer sorting. Therefore coding and decoding require linear time both for Happy and MHappy algorithms. φ_m modifies only edges on the path between 1 and 0, so it preserves the topology of T better than φ_b: this improves the level of locality and heritability of this code.

6 Dandelion code

In the following we present the *Dandelion* code as introduced in [10] with labels on edges:

Dandelion Coding Algorithm
Input: Input: an n-tree T
Output: Output: an $(n-2)$-string C
1. Root T in 0
2. **for** $v = n$ **to** 2 **do**
3. $h = p(v)$, $k = p(1)$, delete (v, h), and add $(v, 1)$ with label h
4. **if** a cycle has been created **then** delete $(1, k)$, add $(1, h)$, $label(v, 1) = k$
5. **for** $v = 2$ **to** n **do** $C[v-2] = label(v, 1)$

The name of dandelion for this code derives from the fact that connecting all the vertices to vertex 1, a tree which looks like a dandelion flower is created (see Figure 2e). Analyzing the algorithm, we can see that the only vertices having the outgoing edge labeled with a value different from their original parent are those verifying the test of line 4, let us call them *flying* vertices.
In code C, a position corresponding to a non-flying vertex v merely displays $p(v)$, showing that the algorithm does not add new information if it considers all the vertices. Hence let us restrict our attention to flying vertices.

Lemma 4. *Flying vertices are all vertices v such that $v \in path(1, 0)$ and $v > max(path(v, 0))$.*

Proof. The first condition trivially holds, otherwise cycles cannot be created.

Given $v \in path(1, 0)$, let $m = max(path(v, 0))$. If $m > v$ then m is processed before v by the algorithm, m is directly connected to 1 and it introduces a cycle containing v. When the cycle is broken (line 4), all the vertices in the cycle are excluded from $path(1, 0)$. This implies that in successive steps v can not be a flying vertex.

On the other hand, if $v > m$ it will be in $path(1, 0)$ when it is processed by the algorithm and so it obviously introduces a cycle.

When a cycle is broken (line 4) in a flying vertex v, 1 will be connected to h (the old parent of v) and the label of edge $(v, 1)$ becomes k (the old parent of 1). In code C, the position corresponding to v displays the value k. Thus, assigning $p(v) = k$, it is possible to avoid edge labels and to generate C directly from p.

Let us define a function φ_d which, given a tree T, constructs a graph G that considers flying vertices of T in decreasing order. For each flying vertex v, φ_d exchange $p(v)$ and $p(1)$ (see Figure 2f).

Theorem 3. *It is possible to fit Dandelion code into our general scheme when* $x = 0$*,* $y = 1$*, and* $\varphi = \varphi_d$*.*

Proof. $G = \varphi_d(T)$ is a functional digraph corresponding to a function g undefined in 0 (the root) and such that $g(1) = 0$ (edge $(1, 0)$). It is also easy to see that the code generated using φ_d is the same as that using Dandelion Coding Algorithm. Considerations similar to those presented in Theorem 2 for φ_m can be used to prove that φ_d is invertible; note that cycles of G must be considered in increasing order of their maximum vertex.

The complexity of the original Dandelion algorithms for coding and decoding is non linear, while it becomes linear when fitted into our general scheme, in view of the fact that φ_d requires the same operations as φ_m.

Concerning locality and heritability, Dandelion code has a behavior which is identical to MHappy code, in spite of the fact that in [10] it was introduced as "a mélange of the methods for Happy code and Blob code". Indeed, both φ_d and φ_m work only on edges in the path between 1 and 0. On this path vertices that modify their parent are maximal (i.e. all vertices v such that $v > max(path(1, v))$) for MHappy code and flying (i.e. all vertices v such that $v > max(path(v, 0))$) for Dandelion code.

7 Conclusion and open problems

In the introduction, we stated that we where looking for codes satisfying properties like efficiency, locality and heritability. Code that is built by associating to each vertex its parent (naïve code), naturally satisfies these properties, but it does not define a bijection between trees and strings.

In this paper we have presented a general scheme for defining bijective codes based on the transformation of a tree into a functional digraph through a function φ. We have emphasized that the required properties are satisfied by our general scheme to the extent that function φ preserves the tree.

The value of our approach is that it returns the characteristics of naïve code as much as is possible: the degree to which function φ preserves the topology of the tree is precisely the same as the degree of similarity between the string obtained using function φ and the string obtained using naïve code. Hence, the required properties are satisfied to this same degree.

We have defined three functions φ_b, φ_m, and φ_d that allow us to fit into our general scheme three known codes: Blob, Happy, and Dandelion codes.

For each of these codes we have shown that it is possible to code and decode in linear time, achieving maximum efficiency. Regarding locality and heritability, we have shown that Happy and Dandelion codes have a performance which is better than Blob code, since they generate functional digraphs with a topology that is very similar to original trees.

Since these codes seem to be suitable candidates for use in evolutionary algorithms, it will be interesting to verify their performance experimentally in tests similar to those reported in [6, 7].

Another interesting view point on these algorithms could be their implementation in a parallel setting: does an efficient parallel way to code and decode trees with high locality and heritability exists?

References

1. Cayley, A.: A theorem on trees. *Quarterly Journal of Mathematics*, 23, pp. 376–378, 1889.
2. Caminiti, S., Finocchi, I., and Petreschi, R.: A unified approach to coding labeled trees. *Proceedings of the 6th Latin American Symposium on Theoretical Informatics*, LNCS 2976, pp. 339–348, 2004. Accepted for publication on Theoretical Computer Science LATIN 2004 Special Issue.
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C.: *Introduction to algorithms*. McGraw-Hill, 2001.
4. Deo, N. and Micikevicius, P.: Parallel algorithms for computing Prüfer-like codes of labeled trees. *Computer Science Technical Report*, CS-TR-01-06, 2001.
5. Edelson, W. and Gargano, M.L.: Feasible encodings for GA solutions of constrained minimal spanning tree problems. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2000)*, Morgan Kaufmann Publishers, pp. 754, 2000.
6. Gottlieb, J., Raidl, G., Julstrom, B.A., and Rothlauf F.: Prüfer Numbers: A Poor Representation of Spanning Trees for Evolutionary Search. *In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pp. 343-350, 2001.
7. Julstrom, B.A.: The Blob Code: A Better String Coding of Spanning Trees for Evolutionary Search. *In 2001 Genetic and Evolutionary Computation Conference Workshop Program*, pp. 256–261, 2001.
8. Kelmans, A., Pak, I., and Postnikov, A.: Tree and forest volumes of graphs. *DIMACS Technical Report* 2000-03, 2000.
9. Kumar, V., Deo, N., and Kumar, N.: Parallel generation of random trees and connected graphs. *Congressus Numerantium*, 130, pp. 7–18, 1998.
10. Picciotto, S.: *How to encode a tree.* Ph.D. Thesis, University of California, San Diego, 1999.
11. Prüfer, H.: Neuer Beweis eines Satzes über Permutationen. *Archiv für Mathematik und Physik*, 27, pp. 142–144, 1918.
12. Zhou, G. and Gen, M.: A note on genetic algorithms for degree-constrained spanning tree problems. *Networks*, 30(2), pp. 91–95, 1997.