versão impressa ISSN 0101-7438 / versão online ISSN 1678-5142

A COMPACT CODE FOR *k*-TREES

Lilian Markenzon*
Núcleo de Computação Eletrônica
Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – RJ, Brazil
markenzon@nce.ufrj.br

Oswaldo Vernet
Núcleo de Computação Eletrônica
Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – RJ, Brazil
oswaldo@nce.ufrj.br

Paulo Renato da Costa Pereira
Instituto Militar de Engenharia (IME)
Rio de Janeiro – RJ, Brazil
prenato@ime.eb.br

* *Corresponding author* / autor para quem as correspondências devem ser encaminhadas

Recebido em 11/2008; aceito em 04/2009
Received November 2008; accepted April 2009

Abstract

In this paper, we propose a new representation for *k*-trees – the *compact code*, which reduces the required memory space from $O(nk)$ to $O(n)$. The encoding and decoding algorithms, based on a simplification of a priority queue, are linear and very simple. As long as the *k*-tree is represented by its compact code, the exact vertex coloring problem can be solved in time $O(n)$.

Keywords: *k*-trees; Prüfer code; compact code.

Resumo

Neste artigo, propomos uma nova representação para *k*-árvores – o *código compacto*, que reduz o espaço de memória de armazenamento de $O(nk)$ para $O(n)$. Os algoritmos de codificação e decodificação, baseados em uma simplificação de uma lista de prioridades, são lineares e muito simples. Uma vez que a *k*-árvore esteja representada pelo seu código compacto, o problema da coloração exata de vértices pode ser resolvido em tempo $O(n)$.

Palavras-chave: *k*-árvores; código de Prüfer; código compacto.

1. Introduction

The family of k-trees, introduced by Harary & Palmer (1968) and revisited somewhat later by Rose (1974), has an inductive definition which naturally extends the definition of a tree. As an important subclass of chordal graphs, k-trees have deserved careful attention of many researchers, either related to algorithmic (Markenzon *et al*., 2006; Proskurowski, 1984) or theoretical aspects (Cai & Maffray, 1993; Lotker *et al*., 2006; Rose, 1974; among several others). In Justel & Markenzon (2000), further properties of the Lexicographic Breadth-First Search were analyzed, introducing an efficient algorithm for recognizing k-trees.

The idea of associating *codewords* to the labelled graphs of a specific family is not recent: in Prüfer (1918), a one-to-one correspondence between the set of $(n–2)$-tuples of the integers $\{1, 2, ..., n\}$ and the set of all labelled 1-trees on n vertices was already proved to hold. A codeword must of course univocally represent a certain graph belonging to the family and, conversely, each graph of the family must be assigned a unique codeword. Besides sparing computer storage significantly, a well designed encoding scheme may lead to more efficient algorithms for solving algorithmic problems for the family.

Rényi & Rényi (1970) extended Prüfer's original codification to a broader family: the labelled rooted k-trees. Initially, the authors defined an immediate extension, the *primitive Prüfer code*, which applied only if the k-tree had the k highest numbered vertices as its root-clique. This constraint was finally eliminated, yielding the *redundant Prüfer code*. Later, Chen (1993) proposed a smaller code for k-trees, based on an intermediate representation using doubly labelled trees. We proposed, in Pereira *et al*. (2005), an $O(n)$-space representation, called the *reduced Prüfer code*, which is equivalent in size to Chen's code and, in Pereira *et al*. (2005a), we provided the first attempts to develop linear time algorithms for this code. Recently, Caminiti *et al*. (2007) presented a new approach for the same problem, using an intermediate code to obtain their results.

The usual representation of k-trees through adjacency lists requires $O(nk)$ memory space. In this paper, an $O(n)$-space representation is proposed, called the *compact code*, which is equivalent in size to Chen's code but much easier to compute. Its applicability is illustrated in Section 7, were the exact vertex coloring problem is solved. The solution presented can be implemented by an algorithm with worst-case time complexity of $O(n)$, in contrast to the $O(m)$ traditional one.

2. Basic Concepts on Chordal Graphs

Let $G = (V, E)$ be a graph, where $| E | = m$ and $| V | = n > 0$. The set of neighbors of a vertex $v \in V$ is denoted as $Adj_G(v) = \{w \in V \mid \{v, w\} \in E\}$. A vertex v is said to be *simplicial* in G when $Adj_G(v)$ is a *clique* in G, i.e., a subset of V that induces a complete subgraph of G.

A *perfect elimination ordering* (*peo*) of $G = (V, E)$ is a bijective function $\sigma:\{1, ..., n\} \rightarrow V$ such that, for $1 \leq i < n$, $\sigma(i)$ is a simplicial vertex in the induced subgraph $G_i = G[\{\sigma(i), ..., \sigma(n)\}]$. A *peo* is ultimately an arrangement of V in a sequence $\sigma(V) = [\sigma(1), ..., \sigma(n)]$, which will be shortly denoted as $\sigma(V) = [v_1, ..., v_n]$. The position of vertex v in σ is given by $\sigma^{-1}(v)$. It is well known that a graph admits a *peo* if, and only if, it is *chordal*, i.e., has no chordless cycle with length greater than 3. Moveover, a given chordal graph G may admit several *peos* (Chandran *et al*., 2003; Golumbic, 2004).

The following lemma establishes basic properties concerning chordal graphs and simplicial vertices.

Lemma 1. (Blair & Peyton, 1993) In a graph $G = (V, E)$, $v \in V$ is simplicial if, and only if, v belongs to exactly one maximal clique.

Once a *peo* $\sigma(V) = [\sigma(i), ..., \sigma(n)]$ is established for a chordal graph $G = (V, E)$, it is possible to define, for each vertex $v \in V$, the set $X_\sigma(v) = \{w \in Adj_G(v) \mid \sigma^{-1}(w) > \sigma^{-1}(v)\}$, called the *monotone adjacency set* of v. In Theorem 2, an interesting result is shown concerning these sets, which will be important to the further development of this paper.

Theorem 2. Let $G = (V, E)$, $n > 0$, be a connected chordal graph and σ, a *peo* of G. For each i such that $1 \leq i < n$, there exists a unique j satisfying: $i < j \leq n$, $v_j \in X_\sigma(v_i)$ and $X_\sigma(v_i) \subseteq \{v_j\} \cup X_\sigma(v_j)$.

Proof. The set $I_i = \{\sigma^{-1}(w) \mid w \in X_\sigma(v_i)\}$ contains all feasible values for j, since the condition $v_j \in X_\sigma(v_i)$ must be satisfied. Take $j = \min I_i$ and let $w \in X_\sigma(v_i)$. We must prove that $w \in \{v_j\} \cup X_\sigma(v_j)$.

- If $w = v_j$, the conclusion is trivial;
- If $w \neq v_j$, as $X_\sigma(v_i)$ is a clique and $v_j \in X_\sigma(v_i)$, $w \in Adj_G(v_j)$. Since $j = \min I_i$, $\sigma^{-1}(w) > j = \sigma^{-1}(v_j)$. Hence, $w \in X_\sigma(v_j)$.

For any other value of j belonging to I_i the condition $X_\sigma(v_i) \subseteq \{v_j\} \cup X_\sigma(v_j)$ does not hold. Thus, j is unique. □

3. Redundant Prüfer Code for *k*-Trees

In Rényi & Rényi (1970), the redundant Prüfer code for *k*-trees was defined and used by the authors in the counting of labelled *k*-trees. In this section, we summarize some results about *k*-trees that allow us to present later an efficient code for this family.

A well known subclass of chordal graphs, the *k*-trees, $k > 0$, can be inductively defined as follows:

- Every complete graph with k vertices is a *k*-tree;
- If $G = (V, E)$ is a *k*-tree, $v \notin V$ and $Q \subseteq V$ is a *k*-clique of G, then $G' = (V \cup \{v\}, E \cup \{\{v, w\} \mid w \in Q\})$ is also a *k*-tree;
- Nothing else is a *k*-tree.

A *k*-tree G with $n > k$ vertices have exactly $n{-}k$ maximal cliques, each of them with k+1 vertices. Moreover, simplicial vertices in *k*-trees with $n > k$ have degree k and are so called *k-leaves*. It can be proved that any maximal clique of G has at most one *k*-leaf. The number of simplicial vertices in a *k*-tree has an interesting behavior: if $n = k$ or $n = k+1$, every vertex is simplicial; for $n > k+1$, there are at least 2 (as in every chordal graph) and at most $n{-}k$ simplicial vertices. *k*-trees can be recognized through the lexicographic breadth-first search in time $O(m)$, by examining the sizes of the labels associated to the vertices during the search (Justel & Markenzon, 2000).

From the aforementioned concepts, it is straightforward to derive a procedure for constructing a *peo* for a *k*-tree G: starting with the empty sequence, at each step, choose a simplicial vertex of G, append it to the sequence and remove it from G (along with its incident edges). When $V = \{1, ..., n\}$, $n > 0$, (or any other totally ordered set), it is possible to particularize this procedure in order to obtain a unique *peo*: at each step, the *least* simplicial vertex is chosen. Since this process of successively removing the least simplicial vertex is essentially the same one employed in Prüfer's algorithm for coding labelled trees, the obtained *peo* will be called the *Prüfer peo* (*ppeo*) of G.

Being σ the *ppeo* of a k-tree $G = (\{1, ..., n\}, E)$, we define the *redundant Prüfer code* of G the sequence of pairs

$$PC(G) = [(v_i, X(v_i)) \mid i = 1, ..., n].$$

The redundant Prüfer code of k-trees has several peculiarities, summarized in Lemma 3, whose proof is straightforward.

Lemma 3. Let $G = (\{1, ..., n\}, E)$ be a k-tree, $n > k$, and $PC(G) = [(v_i, X(v_i)) \mid i = 1, ..., n]$ its redundant Prüfer code. The following properties hold:

1. $| X(v_i) | = k$, $i = 1, ..., n-k$;
2. $| X(v_i) | = n-i$, $i = n-k+1, ..., n$;
3. $\{v_i\} \cup X(v_i)$, $i = 1, ..., n-k$, are the maximal cliques of G;
4. $\{v_i\} \cup X(v_i) = X(v_{i-1})$, $i = n-k+1, ..., n$.

The set $\{v_{n-k}\} \cup X(v_{n-k})$ is a maximal clique of G (hence, with size k+1) called the *residual clique* of G. It will be denoted $\xi(G)$.

Consider, for example, the 3-tree with 10 vertices depicted in Fig.1. Its redundant Prüfer code is

$$\begin{aligned} PC = {} & [(3, \{5,6,9\}),\ (4, \{2,9,10\}),\ (5, \{6,9,10\}),\ (7, \{2,9,10\}),\ (8, \{1,2,6\}), \\ & (1, \{2,6,10\}),\ (2, \{6,9,10\}),\ (6,\{9,10\}),\ (9, \{10\}),\ (10, \emptyset)]. \end{aligned}$$

The residual clique $\xi(G) = \{2, 6, 9, 10\}$ has size k+1 = 4.

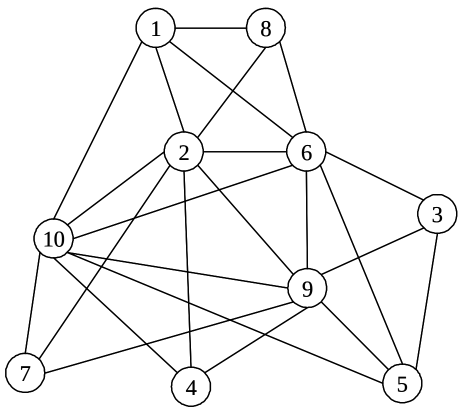

Fig. 1 – A 3-tree with vertices {1, ..., 10}.

The properties stated in Lemma 3 allow us to particularize Theorem 2, from the previous section, to k-trees.

Theorem 4. Let $G = (\{1, ..., n\}, E)$, $n > k$, be a k-tree and $PC(G) = [(v_i, X(v_i)) \mid i = 1, ..., n]$ its redundant Prüfer code. For each i such that $1 \le i \le n-k$, there exists a unique j satisfying $i < j \le n-k+1$, $v_j \in X(v_i)$ and $X(v_i) \subseteq \{v_j\} \cup X(v_j)$.

Corollary 5. Let i and j be indices related to each other according to Theorem 4. Thus,

$$\left| X(v_j) - X(v_i) \right| = \begin{cases} 1, & \text{if } j \le n-k \\ 0, & \text{if } j = n-k+1 \end{cases} \quad \text{and} \quad X(v_i) - X(v_j) = \{v_j\}.$$

4. The Compact Code for k-Trees

The redundant Prüfer code of a k-tree demands half of the number of symbols required by the traditional representation through adjacency sets. However, the occupied space is $O(m)$ in both cases, which means ultimately $O(nk)$. In this section we show how an $O(n)$-space representation can be obtained for this family of graphs, no matter the value of k.

The *compact code* of a k-tree $G = (\{1, ..., n\}, E)$, $n > k$, is the ordered pair $CC(G) = (\xi(G), S)$, where $\xi(G)$ is the residual clique of G and $S = [(\alpha_1, \beta_1), (\alpha_2, \beta_2), ..., (\alpha_{n-k-1}, \beta_{n-k-1})]$ is a sequence of pairs of vertices determined as follows:

```
For $i = 1, 2, ..., n-k-1$:
    Let $j$ be the index corresponding to $i$, according to Theorem 4;
        – $\alpha_i = v_j$;
        – $\beta_i = v \in X(v_j) - X(v_i)$, if $j \le n-k$, or $\beta_i = 0$, if $j = n-k+1$.
```

From Corollary 5, if $j \le n-k$, $\mid X(v_j) - X(v_i) \mid = 1$ and β_i is the unique vertex belonging to $X(v_j) - X(v_i)$; if $j = n-k+1$, the result of the set difference is empty and β_i is set to 0 (a value that does not correspond to any valid vertex).

Since $\mid \xi(G) \mid = k+1$ and $\mid S \mid = n-k+1$, the values of n and k can be deduced from $\mid \xi(G) \mid$ and $\mid S \mid$ and do not need to appear explicitly in the compact code. Notice that $CC(G)$ is not defined for k-trees with $n = k$ and, if $n = k+1$, $CC(G) = (V, [\ ])$.

Applying the definition, the compact code of the 3-tree in Fig. 1 is:

$$CC(G) = (\{2,6,9,10\},\ [(5,10), (2,6), (6,0), (2,6), (1,10), (2,9)]).$$

5. The Encoding Algorithm

The process of obtaining the compact code $CC(G)$ for a given k-tree $G = (\{1, ..., n\}, E)$ is called *encoding* and can be accomplished in two steps:

Step 1. Compute $PC(G) = [(v_i, X(v_i)) \mid i = 1, ..., n]$;

Step 2. Compute $CC(G) = (\xi(G), S)$, using the definition.

In order to develop an algorithm to perform the encoding process, we must give a special attention to *Step 1*, since *Step 2* consists simply in the application of the definition. Given a *k*-tree $G = (\{1, ..., n\}, E)$, the algorithm for obtaining its redundant Prüfer code removes successively from G the least *k*-leaf and appends to the code the vertex and the clique formed by its neighbors.

Some implementation issues must be addressed in order to obtain an efficient algorithm. The main difficult is the determination of the *ppeo*, i.e., the computation, at each step, of the least *k*-leaf. A traditional priority queue can be used to store the *k*-leaves of graph G and the vertices that become simplicial during the process, providing a way to identify the least numbered one. However, the time complexity of employing this data structure is $O(n \log n)$.

Based on the specificity of the problem, we are able to define a particular case of priority queue, which is similar but even simpler than the (L,U)-bounded priority queue presented in Markenzon *et al*. (2006) as the elements and their priorities coincide.

Since the *k*-leaves are recognized by their degree (a vertex v is a *k*-leaf if $degree(v) = k$), the only data structure needed to implement the priority queue is the array that stores this information. At the first iteration, the algorithm searches this array sequentially looking for the least *k*-leaf of the current graph $G_1 = G$, which is then stored in the variable *last*. Vertex v_i, $i = 1$, is determined. At the end of the *i-th* iteration, v_i is removed from G_i, yielding G_{i+1}. The array *degree* must be updated (this updating is equivalent to an *Insert* operation). The maximal clique $\{v_i\} \cup X(v_i)$ disappears from G_i and any vertex of $X(v_i)$ may become a simplicial vertex. However, as we know that any maximal clique of a *k*-tree G has at most one simplicial vertex, only one new *k*-leaf w can appear in G_{i+1}. Two cases can happen:

case 1: $w > last$, that is, w is not the least simplicial vertex of G_{i+1};

case 2: $w < last$. In this case, w must be the next simplicial vertex chosen.

Differently than for case 1, for the second case a new variable must be defined, completing the implementation of the priority queue. The variable *pend* stores w which will be the next mandatory simplicial vertex chosen. If there is a pending element, it is simply removed; otherwise, the next element with lowest priority is sought and removed. This computation is repeated until it remains only the residual clique in the graph. At this point, a simple ordering of its elements completes the *ppeo*.

Graph G, represented by its adjacency lists, is the input of the algorithm.

```
Procedure $DeleteMin(v)$;
    if $pend > 0$ then
        $v \leftarrow pend$;   $pend \leftarrow 0$;
    else
        while $degree[last] \neq k$ do $last \leftarrow last + 1$;
        $v \leftarrow last$;
    $degree(v) \leftarrow 0$;

Procedure $Encode\text{-}Step1$;
    for $u \leftarrow 1, ..., n$ do $degree(u) \leftarrow |Adj_G(u)|$;
    $pend \leftarrow 0$;   $last \leftarrow 1$;
    for $i \leftarrow 1, ..., n-k-1$ do
        $DeleteMin(v_i)$;
        $X(v_i) \leftarrow \{w \in Adj_G(v_i) \mid degree(w) \neq 0\}$;
        for all $w \in X(v_i)$ do
            degree(w) $\leftarrow$ degree(w) $- 1$;
            if ($degree(w) = k$) and ($w < last$) then $pend \leftarrow w$;
        $pend \leftarrow 0$;   $last \leftarrow 1$;
        for  $i \leftarrow n-k, ..., n$ do
            $DeleteMin(v_i)$;
            $X(v_i) \leftarrow \{w \in Adj_G(v_i) \mid degree(w) \neq 0\}$;

    Procedure Encode-Step2;
        for  $i \leftarrow 1, ..., n$ do $\sigma^{-1}(v_i) = i$;  $S \leftarrow [\ ]$;
        for  $i \leftarrow 1, ..., n-k-1$ do
            $j \leftarrow \min\{\sigma^{-1}(u) \mid u \in X(v_i)\}$;
            $\alpha_i \leftarrow v_j$;   $\beta_i \leftarrow 0$;
            if  $j \neq n-k+1$ then  $\beta_i \leftarrow X(v_j) - X(v_i)$;
            $S \leftarrow S \,||\, (\alpha_i, \beta_i)$;
        $CC(G) \leftarrow (\{v_{n-k}, ..., v_n\}, S)$;
```

The encoding algorithm is linear, as shown below.

Procedure *Encode-Step1* is linear. The initialization of the array *degree* takes $O(m)$. At each iteration the least simplicial vertex is chosen. During the *DeleteMin* operation two situations may arise: if there is a pending element, it is simply removed; otherwise, the next element with lowest priority is sought and removed. The first one takes $O(1)$; the second, an overall complexity of $O(n)$. The deletion of a simplicial vertex v is $O(1)$; the vertices of $Adj(v)$ that do not appeared yet at the *ppeo* constitute the monotone adjacency set $X(v)$. So, the overall complexity is $O(m)$. Procedure *Encode-Step2* is also linear. In the proof of Theorem 2, we show that j is the first vertex of $X(v_i)$ that appears in the *ppeo*. At each iteration, $\min\{\sigma^{-1}(u) \mid u \in X(v_i)\}$ is determined. So, it takes $\sum_1^{n-k-1} X(v_i) = O(m)$.

6. The Decoding Algorithm

The inverse task is called *decoding*: given the pair (ξ, S), being $|S| > 0$, the redundant code of G, $PC(G) = [(v_i, X(v_i)) \mid i = 1, ..., n]$, must be determined. We present three lemmas that support the decoding algorithm.

Lemma 6. Let G be a k-tree, $n > k+1$, $CC(G)$ its compact code and $\sigma = [v_1, ..., v_n]$ its *ppeo*. For $i = n-k-1, ..., 1$, if $\beta_i = 0$ then $X(v_i) = \xi - \{v_{n-k}\}$ else $X(v_i) = \{\alpha_i\} \cup X(\alpha_i) - \{\beta_i\}$.

Proof. By definition, if $j = n-k+1$ then $\beta_i = 0$. From Theorem 4, $X(v_i) = \{v_{n-k+1}\} \cup X(v_{n-k+1}) = X(v_{n-k})$. Also by definition, if $\beta_i \neq 0$, $\alpha_i = v_j$ and $\{\beta_i\} = X(\alpha_i) - X(v_i)$. Since $\alpha_i \notin X(\alpha_i)$, then $X(v_i) = \{\alpha_i\} \cup X(\alpha_i) - \{\beta_i\}$. □

Lemma 7. Let G be a k-tree, $n > k+1$ and $CC(G)$ its compact code. For $i = 1, ..., n-k-1$:

$$\bigcup_{t=i}^{n} X(v_t) = \bigcup_{t=i}^{n-k-1} \{\alpha_t\} \cup \xi .$$

Proof. By induction on the size of S.

- Let $t = n–k–1$. By Lemma 3, $X(v_{n-k-1}) \subset \{v_{n-k}\} \cup X(v_{n-k})$. By definition, $\alpha_{n-k-1} = v_{n-k}$ or $\alpha_{n-k-1} = v_{n-k+1}$, i.e., $\alpha_{n-k-1} \in \xi$. Then S has one pair and

$$\bigcup_{t=n-k-1}^{n} X(v_t) = \{\alpha_{n-k-1}\} \cup \xi \,.$$

- Suppose that the lemma is valid for i+1, $1 \le i < n–k–1$. Then

$$\bigcup_{t=i+1}^{n} X(v_t) = \bigcup_{t=i+1}^{n-k-1} \{\alpha_t\} \cup \xi \,. \qquad (1)$$

- From Lemma 6, (i) $X(v_i) = \{\alpha_i\} \cup X(\alpha_i) - \{\beta_i\}$ or (ii) $X(v_i) = X(v_{n-k})$ and $\alpha_i = v_{n-k+1} \in X(v_{n-k+2})$. Then

$$\bigcup_{t=i}^{n} X(v_t) = X(v_i) \cup \left(\bigcup_{t=i+1}^{n} X(v_t)\right) = \{\alpha_i\} \cup \left(\bigcup_{t=i+1}^{n} X(v_t)\right).$$

By (1),

$$\bigcup_{t=i}^{n} X(v_t) = \{\alpha_i\} \cup \left(\bigcup_{t=i+1}^{n-k-1} \{\alpha_t\}\right) \cup \xi \,. \quad \square$$

Lemma 8. The vertices v_i, $i = 1, ..., n–k–1$ are determined as follows:

$$v_i = \min\left(V - \xi - \bigcup_{t=i}^{n-k-1} \{\alpha_t\} - \bigcup_{t=1}^{i-1} \{v_t\}\right).$$

Proof. By definition, v_i is the least simplicial vertex in the induced subgraph $G_i = G[\{v_i, v_{i+1}, ..., v_n\}]$. Then $v_i \notin V - \bigcup_{t=1}^{i-1}\{v_t\}$ and $v_i \notin \bigcup_{t=i}^{n} X(v_t) = \bigcup_{t=i}^{n-k-1}\{\alpha_t\} \cup \xi$. $\square$

The decoding algorithm consists of the following steps:

Step 1. Set $k = |\xi| - 1$ and $n = |S| + |\xi|$;

Step 2. Sort set ξ ascendingly obtaining the pairs $(v_{n-k}, X(v_{n-k}))$, ..., $(v_n, \varnothing)$;

Step 3. Compute $[v_1, ..., v_{n-k-1}]$, subsequence of σ;

Step 4. Compute $X(v_i)$, for $i = n–k–1, ..., 1$.

Step 1 is trivial. *Step 2* computes the pairs originated from the residual clique.

The third step computes $[v_1, ..., v_{n-k-1}]$, based on Lemma 8. The set of feasible vertices must be determined (and updated) for the choice of the minimum. The terms ξ and $\bigcup_{t=i}^{i-1}\{v_t\}$ are well known. The set $\bigcup_{t=i}^{n-k-1}\{\alpha_t\}$ is actually a multiset and its composition is settled by the number of times that a vertex appears as a label in the sequence S. So, the frequency of each vertex must be stored and updated at each iteration of the algorithm. The implementation of *Step 3* uses the same simplified priority queue mentioned in Section 5.

After having obtained the *ppeo*, the monotone adjacency sets $X(v_i)$, $i = n–k–1, ..., 1$, must be determined (*Step 4*). Lemma 6 shows how to compute these sets. Notice that the *ppeo* is explored in the reverse ordering.

The decoding algorithm is also linear; the proof is similar to the one presented for the encoding algorithm.

7. An Application

Besides providing a more concise representation for a *k*-tree, the compact code can support better solutions for some problems. A straightforward example is the determination of the *ppeo*, whose solution can be achieved in time $O(n)$, according to the results presented in Section 6. Another example is provided in this section.

Exact vertex coloring is a very important problem in algorithmic graph theory, which can be solved through polynomial algorithms only for some particular families of graphs. The problem consists in associating colors to the vertices of a graph, so that the endpoints of each edge receive distinct colors, using the minimum number of colors.

In particular, every chordal graph G can be exactly colored in time $O(m)$ using $\omega(G)$ distinct colors, where $\omega(G)$ is the size of the largest clique in the graph. Due to this strong result, no further efforts are usually made to solve the problem for its subclasses. We show here a solution for a *k*-tree represented by its compact code in time $O(n)$.

If a *k*-tree $G = (\{1, ..., n\}, E)$, $n > k$, is represented by its compact code, the residual clique $\xi(G)$ has size k+1 and demands exactly (k+1) colors. The remaining vertices can be colored following Lemma 9.

Lemma 9. Let $G = (\{1, ..., n\}, E)$, $n > k$, be a *k*-tree, $CC(G) = (\xi(G), S)$ its compact code and $\sigma = [v_1, ..., v_n]$ its *ppeo*. Assume that the vertices $v_{n–k}, ..., v_n$ are already colored with k+1 distinct colors. Thus, for $i = n–k–1, ..., 1$:

$$color(v_i) = \begin{cases} color(v_{n-k}) & \text{if } \beta_i = 0; \\ color(\beta_i) & \text{otherwise.} \end{cases}$$

An $O(n)$-time procedure can easily implement the solution, in contrast to the corresponding $O(m)$-time algorithm for chordal graphs.

Acknowledgement

This work was supported by grants 306893/2006-1 and 473603/2007-1, CNPq, Brazil.

References

(1) Blair, J.R.S. & Peyton, B. (1993). An Introduction to Chordal Graphs and Clique Trees. *Graph Theory and Sparse Matrix Computation*, IMA **56**, 1-29.

(2) Chen, W.Y.C. (1993). A Coding Algorithm for Rényi Trees. *Journal of Combinatorial Theory*, series A, **63**, 11-25.

(3) Chandran, L.S.; Ibarra, L.; Ruskey, F. & Sawada, J. (2003). Generating and Characterizing the Perfect Elimination Orderings of a Chordal Graph. *Theoretical Computer Science*, **307**, 303-317.

(4) Cai, L. & Maffray, F. (1993). On the Spanning *k*-Tree Problem. *Discrete Applied Mathematics*, **44**, 139-156.

(5) Caminiti, S., Fusco, E.G. & Petreschi, R. (2007). A Bijective Code for *k*-Trees with Linear Time Encoding and Decoding. *LNCS*, **4614**, 408-420.

(6) Golumbic, M.C. (2004). *Algorithmic Graph Theory and Perfect Graphs*. 2^{nd} edition. Academic Press, New York.

(7) Harary, F. & Palmer, E.M. (1968). On Acyclic Complexes. *Mathematika*, **15**, 115-122.

(8) Justel. C.M. & Markenzon, L. (2000). Lexicographic Breadth First Search and *k*-Trees. *Proc. of JIM'2000 – Secondes Journées de l'Informatique Messine*, 23-28, Metz, France.

(9) Lotker, Z.; Majumdar, D.; Narayanaswamy, N.S. & Weber, I. (2006). Sequences Characterizing *k*-Trees. *Proc. of COCOON'06, LNCS*, **4112**, 216-225.

(10) Markenzon, L.; Justel, C.M. & Paciornik, N. (2006). Subclasses of *k*-Trees: Characterization and Recognition. *Discrete Applied Mathematics*, **154**, 818-825.

(11) Markenzon, L.; Vernet, O. & Pereira, P.R.C. (2006). (*L*,*U*)-Bounded Priority Queues. *Proc. of CLAIO'06*.

(12) Proskurowski, A. (1984). Separating Subgraphs in *k*-Trees: Cables and Caterpillars. *Discrete Mathematics*, **49**, 275-285.

(13) Prüfer, A. (1918). Neuer Beweis eines Satzes über Permutationen. *Archiv der Mathematik und Physik*, **27**, 142-144.

(14) Pereira, P.R.C.; Markenzon, L. & Vernet, O. (2005). The Reduced Prüfer Code for Rooted Labelled *k*-Trees. *Electronic Notes in Discrete Mathematics*, **22**, 135-139.

(15) Pereira, P.R.C.; Markenzon, L. & Vernet, O. (2005a). Código Reduzido de Prüfer para *k*-Árvores Rotuladas. *Anais do XXXVI Simpósio Brasileiro de Pesquisa Operacional*, 2335-2342 (in portuguese).

(16) Rose, D.J. (1974). On Simple Characterizations of *k*-Trees. *Discrete Mathematics*, **7**, 317-322.

(17) Rényi, C. & Rényi, A. (1970). Prüfer Code for *k*-Trees. **In**: *Combinatorial Theory and its Applications* [edited by P. Erdös *et al.*], 945-971.