(L, U)-Bounded Priority Queues and the Codification of Rényi k-Trees

Lilian Markenzon[1] *, Oswaldo Vernet[1] and Paulo Renato C. Pereira[2]

[1] Núcleo de Computação Eletrônica
Universidade federal do Rio de Janeiro
{markenzon,oswaldo}@nce.ufrj.br
[2] Instituto Militar de Engenharia
pprrl@ime.eb.br

Abstract. We introduce in this paper a data structure named (L, U)-*bounded priority queue*, which particularizes priority queues in two aspects: the priorities associated to the elements must be integer numbers constrained to a predefined interval and, in a sequence of operations, no more than one $Insert$ can immediatly follow a $DeleteMin$. This data structure is used in the development of efficient algorithms for coding and decoding Rényi k-trees.

1 Introduction

The family of k-trees, introduced by Harary and Palmer [7] and revisited somewhat later in [11], 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 in graph theory. Rényi and Rényi [12] extended to k-trees the method whereby Prüfer [9] had coded 1-trees. They also established feasibility conditions, stating necessary and sufficient conditions for a sequence to be the codeword of a k-tree with n vertices. A central point in their paper was determining the number of labelled k-trees with n vertices. In 1993, Chen [3] reapproaches the subject, proposing a more compact code for k-trees, which can be obtained from an intermediate representation using doubly labelled trees. An alternative code for k-trees was also presented in [10], which has the same size as Chen's code but is much simpler to compute.

From an algorithmic viewpoint, some papers must be mentioned, although all of them deal only with the encoding and decoding of 1-trees. In 2000, Chen and Wang [4] stressed that, although the problem of producing a Prüfer code in linear time is an exercise in two books, there exists no explicit publication of a solution, and proposed one. Soon after, Deo and Micikevicius [5] presented a survey on Prüfer-like codes for labelled trees, gathering and classifying several related codification algorithms. Caminiti *et al.* [2] approached the problem in an unified way, providing linear time algorithms for encoding and decoding under several encoding schemes, being the Prüfer code among them. We were not able

* Partially supported by grant 301068/2003-8, CNPq, Brazil.

to find in the literature linear-time algorithms for encoding and decoding k-trees after the encoding scheme proposed by Rényi and Rényi.

In this paper we are interested in the study of such algorithms. A straightforward implementation of the method proposed in [12] takes $O(n \log n)$ time, using ordinary priority queues. However it is well known that, in the development of efficient algorithms, finding the most suitable data structure plays a fundamental role. We propose thus some restrictions on priority queues which lead to the definition of a particular data structure, called (L, U)-*Bounded Priority Queue*.

We present in Section 2 the formal definition of this new data structure along with the amortized analysis of the worst-case time complexity of performing a valid sequence of operations. In Section 3, basic concepts about k-trees are presented, as well as the Rényi k-trees and their coding. In Section 4, we show how to use the new data structure to obtain linear-time algorithms for encoding Rényi k-trees. The decoding procedure is outlined in Section 5. Both algorithms have worst-case time complexity $O(m)$, where m is the number of edges of the Rényi k-tree and, in the special case where $k = 1$, the algorithms have the same time complexity as the best known algorithms for Prüfer coding.

2 (L, U)-Bounded Priority Queues

A *priority queue* Q is a collection of elements, each of them having an integer priority, on which the following operations are defined:

- $MakeEmptyQueue(Q)$: initializes Q with no elements.
- $Insert(Q, pri, el)$: adds to Q the element el whose priority is pri.
- $DeleteMin(Q, pri, el)$: returns the element with lowest priority stored in Q.

When binary heaps are used to implement priority queues, the worst-case time complexities of these operations are respectively $O(1)$, $O(\log n)$ and $O(\log n)$ [13].

We introduce here a particular case of priority queues, named (L, U)-*bounded priority queues*, to which the following restriction apply: the priorities must be constrained to a predefined range of integer values, being L and U the lowest and highest priority values respectively.

A sequence of operations on an (L, U)-bounded priority queue is considered *valid* when:

- it begins with the $MakeEmptyQueue$ operation;
- the i-th $DeleteMin$ is preceded by at least i $Insert$'s;
- at most one $Insert$ appears immediatly after each $DeleteMin$.

The first two restrictions apply to any data structure, whereas the third one makes the (L, U)-bounded priority queue a special case of priority queues. In this case, a particular implementation allows to make an amortized analysis for a valid sequence of operations, yielding a more realistic upper bound for the execution time.

An (L, U)-bounded priority queue can be represented by a record containing the following informations:

- *vector*: an array over the range $[L..U]$ of linked lists of elements; each position in *vector* corresponds to a feasible value of priority.
- *pend*: an integer variable indicating the priority of a pending element ($pend < L$ means that there is no pending element).
- *last*: an integer variable indicating the priority of the last removed non-pending element.
- *lower*, *upper*: keep the values of L and U, respectively.

The implementation of the fundamental operations is shown next. Consistency checks are omitted for the sake of readability, assuming that the operations are invoked from a valid sequence.

$MakeEmptyQueue$ initializes Q as an empty (L, U)-bounded priority queue.

```
procedure MakeEmptyQueue(Q, L, U);
begin
    for i ← L, ..., U do Q.vector[i] ← ∅;
    Q.pend ← L − 1;
    Q.last ← Q.lower ← L;
    Q.upper ← U
end.
```

The execution of an $Insert$ operation appends a new element to the structure. If the priority of this element is greater than or equal to the priority of the last removed element, a new node is simply added to the list of the corresponding priority; otherwise, a pending element is generated, which will be the next to be removed. Since no more than one $Insert$ can appear immediatly after a $DeleteMin$ in a valid sequence of operations, there will be at most one pending element at any given time.

```
procedure Insert(Q, pri, el);
begin
    InsertList(Q.vector[pri], el);
    if pri < Q.last then
        Q.pend ← pri
end.
```

During a $DeleteMin$ operation two situations may arrise: if there is a pending element, it is simply removed; otherwise, the next element with lowest priority is sought and removed.

```
procedure DeleteMin(Q, pri, el);
begin
    if Q.pend ≥ Q.lower then
        pri ← Q.pend;
        Q.pend ← Q.lower − 1
    else
        while Q.vector[Q.last] = ∅ do
            Q.last ← Q.last + 1;
        pri ← Q.last;
    el ← RemoveList(Q.vector[pri])
end.
```

Being $\Delta = U - L + 1$, the individual worst-case time complexities of the operations are $O(\Delta)$, $O(1)$ and $O(\Delta)$, respectively. In Theorem 1, an amortized analysis allows us to evaluate the worst-case time complexity of performing a valid sequence of operations on an (L, U)-bounded priority queue.

Theorem 1. *Any valid sequence containing exactly t $Insert$ operations on an (L, U)-bounded priority queue can be executed, in the worst case, in time $O(t + \Delta)$, where $\Delta = U - L + 1$.*

Proof. The initial $MakeEmptySet$ operation is performed in time $O(\Delta)$, whereas $Insert$ operations take $O(1)$ time. Let d be the total number of $DeleteMin$ operations. The j-th $DeleteMin$ takes time $O(\omega_j - \alpha_j + 1)$, where α_j and ω_j are respectively the values of the variable *last* before and after the operation is executed. Thus, the total time is

$$O(\Delta) + \sum_{j=1}^{t} O(1) + \sum_{j=1}^{d} O(\omega_j - \alpha_j + 1).$$

Since $\alpha_{j+1} = \omega_j$, for $j = 1, ..., d - 1$, the expression simplifies to

$$O(\Delta + t + d + \omega_d - \alpha_1).$$

As $d \leq t$, $\alpha_1 = L$ and, in the worst case, $\omega_d = U$, the result holds. □

3 Rényi *k*-Trees and Their Coding

Let $G = (V, E)$ be a graph, with $|V| = n$ and $|E| = m$. For any vertex $v \in V$, let $Adj_G(v) = \{w \in V \mid \{v, w\} \in E\}$ be the *set of neighbors* of v. $|Adj_G(v)|$ is the *degree* of v. Given $S \subseteq V$, we denote as $G[S] = (S, \{\{x, y\} \in E \mid x \in S \wedge y \in S\})$ the *subgraph of G induced by S*. S is a clique when $G[S]$ is a complete graph. If $Adj_G(v)$ is a clique in G, then v is said to be a *simplicial* vertex. A *perfect elimination ordering* for G is an ordering $\langle v_1, \ldots, v_n \rangle$ of its n vertices so that v_i is simplicial in $G[\{v_i, \ldots, v_n\}]$, for $i = 1, \ldots n$. *Chordal graphs* are precisely the class of graphs for which a perfect elimination ordering can be obtained.

The following lemmas establish basic properties concerning chordal graphs and simplicial vertices.

Lemma 1 (Dirac 1961, [6]). *Every chordal graph $G = (V, E)$ has a simplicial vertex. Moreover, if G is not a complete graph, then it has two nonadjacent simplicial vertices.*

Lemma 2 ([1]). *In a graph $G = (V, E)$, $v \in V$ is simplicial if, and only if, v belongs to exactly one maximal clique.*

Recall that the *clique-intersection graph* of a graph G is the connected weighted graph whose vertices are the maximal cliques of G and whose edges

connect vertices corresponding to non-disjoint maximal cliques. Each edge is assigned an integer weight, given by the cardinality of the intersection between the maximal cliques represented by its endpoints. Every maximum-weight spanning tree of the clique-intersecton graph of G is called a *clique-tree* of G. If G is chordal, the clique-trees of G obey the *intersection property*: if Q_1 and Q_2 are maximal cliques of G, the intersection $Q_1 \cap Q_2$ is a subset of any maximal clique of G lying on the path between Q_1 and Q_2 in any clique-tree of G. See [1] for more details.

A k-tree, $k > 0$, is a graph that can be inductively defined as follows:

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

It can be proved that, in a k-tree G with n vertices, there are exactly $n - k$ maximal cliques, each of them with $k + 1$ vertices, and the edges of any clique-tree of G have weight k. Moreover, simplicial vertices in k-trees with $n > k$ have degree k and are so called *k-leaves*. The number of simplicial vertices in a k-tree has an interesting behaviour: 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 [8].

Lemma 3. *Let $G = (V, E)$ be a k-tree with $n > k + 1$. Any maximal clique of G has at most one k-leaf.*

Proof. Suppose, to the contrary, that G has a maximal clique C with 2 k-leaves, i.e., simplicial vertices. Since $n > k + 1$ and the edges of any clique-tree of G have weight k, there must exist another maximal clique C' with k vertices in common with C. Hence, one of the simplicial vertices should also belong to C', contradicting the result of Lemma 2. □

A k-tree $G = (V, E)$, where $V = \{1, \ldots, n\}$, $n > k$ and whose k highest vertices form a k-clique is called a *Rényi k-Tree*. The k-clique $\{n - k + 1, \ldots, n\}$ is the *root clique* of G, or, equivalently, G is said to be *rooted* at this k-clique. Observe that the vertices coincide with their labels and that such k-trees are not defined for $n = k$.

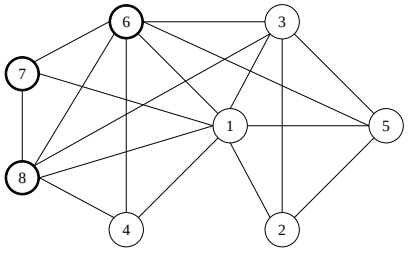
The *redundant Rényi code* ([12]) for a Rényi k-tree $G = (V, E)$ is the array

$$\begin{pmatrix} a_1 & a_2 & \ldots & a_{n-k} \\ B_1 & B_2 & \ldots & B_{n-k} \end{pmatrix}$$

where a_i are vertices and B_i are k-cliques of G, for $i = 1, \ldots, n - k$, obtained through the following algorithm:

$G_1 \leftarrow G$;
for $i \leftarrow 1, \ldots, n-k$ **do**
  Let a_i be the least numbered k-leaf of G_i
    not belonging to the root clique;
  $B_i \leftarrow Adj_{G_i}(a_i)$;
  $G_{i+1} \leftarrow G_i - a_i$;

Figure 1 shows a Rényi 3-tree and the determination of its code.



i	a_i	B_i
1	2	$\{1,3,5\}$
2	4	$\{1,6,8\}$
3	5	$\{1,3,6\}$
4	3	$\{1,6,8\}$
5	1	$\{6,7,8\}$

Figure 1: A Rényi 3-tree and its Redundant Rényi Code

Notice that B_{n-k} turns out to be the root clique. Furthermore the $n-k$ maximal cliques of G are exactly $\{a_1\} \cup B_1, \ldots, \{a_{n-k}\} \cup B_{n-k}$ and the sequence $\langle a_1, \ldots, a_{n-k}, n-k+1, \ldots n\rangle$ is a perfect elimination ordering for G. So, the original Rényi k-tree G can be easily reconstructed by processing its redudant code backwards.

The *primitive Rényi code* (or simply the *Rényi code*) of G is the sequence of k-cliques $\mathcal{R}(G) = \langle B_1, \ldots, B_{n-k-1}\rangle$. Therefore, when $n = k+1$, the Rényi code is the empty sequence. When $k = 1$, it is easy to see that the Rényi code coincides with the Prüfer code [9] for the corresponding 1-tree.

In [12], Rényi and Rényi established *admissibility conditions*, which allow to verify whether a given sequence of $n-k-1$ subsets, each of them with k elements from $\{1, \ldots, n\}$, is the code of a Rényi k-tree; if so, it is possible to reconstruct the original Rényi k-tree from the code, as will be shown in Section 5.

4 The Encoding Algorithm

Given a Rényi k-tree $G = (V, E)$, being $V = \{1, \ldots, n\}$, $n > k$ and $|E| = m$, the algorithm for obtaining its redundant and primitive codes removes successively from G the least k-leaf and appends to the code the k-clique formed by its neighbors.

An (L, U)-bounded priority queue Q, where $L = 1$ and $U = n$, can be used to store the initial k-leaves and the vertices that become k-leaves during the process, providing a fast way to identify the least numbered one. In this case, the elements and their priorities coincide. The overhead of explicitly constructing the sequence of graphs G_i, $i = 1, \ldots, n-k+1$, can be avoided by associating to each vertex a counter, that is initialized with the degree of the vertex and is decremented appropriately.

The following procedure determines the redundant code of a Rényi k-tree $G = (V, E)$, being $V = \{1, \ldots, n\}$, $n > k$, represented by adjacency lists $Adj_G(v)$, for all $v \in V$.

```
procedure Encode;
begin
  MakeEmptyQueue(Q, 1, n);
  for v ← 1, ..., n do
    degree(v) ← |Adj_G(v)|;
    if (v ≤ n − k) and (degree(v) = k) then
      Insert(Q, v, v);
  for i ← 1, ..., n − k − 1 do
    DeleteMin(Q, a_i, a_i);
    degree(a_i) ← 0;
    B_i ← ∅;
    for all v ∈ Adj_G(a_i) | degree(v) > k do
      B_i ← B_i ∪ {v};
      degree(v) ← degree(v) − 1;
      if (v ≤ n − k) and (degree(v) = k) then
        Insert(Q, v, v);
  DeleteMin(Q, a_{n−k}, a_{n−k});
  B_{n−k} ← {n − k + 1, ..., n};
end.
```

In order to prove that the sequence of operations on Q performed in the encoding algorithm is valid, we analyze in Lemma 4 how the number of k-leaves in a k-tree can vary when a k-leaf is removed.

Lemma 4. *Let $G = (V, E)$ be a k-tree with $n > k + 2$. By removing a k-leaf from G, at most one new simplicial vertex appears.*

Proof. Let v be a simplicial vertex and $C_v = \{v, \ell_1, \ldots, \ell_k\}$ be the unique maximal clique of G containing v (Lemma 2). $G - v$ is evidently a k-tree with the same maximal cliques as G, except C_v. Moreover, only the vertices $\ell_1, \ldots, \ell_k$, which are adjacent to v in G, can become simplicial in $G - v$, since, by Lemma 3, none of them can be simplicial in G.

So, let us suppose by contradiction that $G-v$ has two new simplicial vertices: ℓ_i and ℓ_j, being $i \neq j$. Since $n > k+2$, by Lemma 2, there exist distinct maximal cliques C_i and C_j in $G - v$ such that ℓ_i belongs only to C_i and ℓ_j belongs only to C_j.

Let T be a clique-tree of G. We claim that C_i and C_v are adjacent in T, otherwise, as $\ell_i \in C_v \cap C_i$, by the intersection property, there should be a maximal clique $C_i^* \neq C_i$ also containing ℓ_i, lying on the path between C_v and C_i in T and adjacent to C_v, contradicting the fact that ℓ_i belongs only to C_i in $G - v$. Likewise, C_j and C_v must also be adjacent in T. Since adjacent cliques in the clique-tree of a k-tree must have k vertices in common, we must have $C_i \cap C_v = C_j \cap C_v = \{\ell_1, \ldots, \ell_k\}$. Thus ℓ_i and ℓ_j belong both to C_i and C_j, contradicting Lemma 2. So, $G - v$ has at most one new simplicial vertex. □

It must be observed that the previous lemma does not hold when $n = k + 2$; in this case, the k-tree has exactly 2 k-leaves and, by removing one of them, a k-tree with $k + 1$ vertices is obtained, all of them being k-leaves.

Lemma 5. *The sequence of operations performed on Q in the encoding algorithm is valid and contains exactly $n - k$ $Insert$ operations.*

Proof. A vertex v is inserted into Q when $degree(v) = k$, as long as it does not belong to the root clique ($v \leq n - k$). During the encoding process, this will occur exactly once for each of the vertices $1, \ldots, n - k$. By Lemma 4, when a k-leaf is removed, at most one new k-leaf is generated in the resultant k-tree. So, at most one $Insert$ will appear immediately after a $DeleteMin$ in the sequence of operations. Moreover, at each iteration, the remaining graph after the removal of a k-leaf is a k-tree and, as so, it has at least two simplicial vertices. Then Q is never empty when a $DeleteMin$ operation is performed. □

The complexity of the encoding procedure is analyzed in Theorem 2.

Theorem 2. *The worst-case time complexity of the encoding algorithm is $O(m)$.*

Proof. By Theorem 1, the time spent in performing the valid sequence of operations on Q is $O(n + n - k) = O(n)$, since the sequence contains $n - k$ $Insert$ operations. However, in the main loop, $n - k - 1$ vertices have their adjacency lists traversed, what may take time $O(m)$. □

5 The Decoding Algorithm

Given n, k, $n > k$, and a sequence $B = \langle B_1, \ldots, B_{n-k-1} \rangle$ of $n - k - 1$ subsets of size k from $\{1, \ldots, n\}$, a relevant question is whether there exists a Rényi k-tree $G = (V, E)$, being $V = \{1, \ldots, n\}$, such that $\mathcal{R}(G) = B$. Theorem 3, proved in [12], gives the necessary and sufficient conditions.

Theorem 3 ([12]). *A sequence $\langle B_1, \ldots, B_{n-k-1} \rangle$ of $n - k - 1$ subsets of size k from $\{1, \ldots, n\}$ is the Rényi code of a k-tree with n vertices, labelled by the numbers $1, \ldots, n$ and rooted at the k-clique $\{n - k + 1, \ldots, n\}$, if and only if it has the following properties:*

- *Putting $B_{n-k} = \{n - k + 1, \ldots, n\}$, denoting a_1 the least natural number not belonging to $B_1 \cup \ldots \cup B_{n-k}$ and, for every i such that $1 < i \leq n - k$, denoting by a_i the least natural number not belonging to $\{a_1\} \cup \ldots \cup \{a_{i-1}\} \cup B_i \cup \ldots \cup B_{n-k}$, the numbers $a_1, \ldots, a_{n-k}$ form a permutation of the numbers $1, \ldots, n - k$.*
- *For each $i = 1, \ldots, n - k - 1$, there exists j such that $i < j \leq n - k$ and $B_i \subset \{a_j\} \cup B_j$.*

Theorem 3 not only gives the admissibility conditions, but also leads to the decoding algorithm as well. This algorithm takes as input the sequence $B = \langle B_1, \ldots, B_{n-k-1} \rangle$ and issues as output the corresponding Rényi k-tree $G = (V, E)$.

The process consists in obtaining the missing elements of the redundant Rényi code $\begin{pmatrix} a_1 & a_2 & \dots & a_{n-k} \\ B_1 & B_2 & \dots & B_{n-k} \end{pmatrix}$, from which the original Rényi k-tree can be easily reconstructed. As $B_{n-k} = \{n-k+1, \dots, n\}$ is the root clique, the elements $a_1, \dots, a_{n-k}$ are to be determined; they correspond exactly to the k-leaves that were removed when the code was constructed.

For each vertex $v \in V$, a counter $freq(v)$ is maintained, whose value at the beginning of the i-th iteration represents how many times v occurs in the sequence of sets $S_i = \langle \{a_1, \dots, a_{i-1}\}, B_i, \dots, B_{n-k} \rangle$. By Theorem 3, the least vertex v for which $freq(v) = 0$ is exactly the element a_j. At the end of the i-th iteration, $freq(a_i) = 1$ and $freq(w)$ is decremented, for all $w \in B_i$.

As an example, let us consider $n = 8$, $k = 3$ and $B = \langle \{1,3,5\}, \{1,6,8\}, \{1,3,6\}, \{1,6,8\} \rangle$ the primitive code obtained for the Rényi k-tree in Figure 1. The decoding process performs the following $n - k = 5$ steps:

i	S_i	$freq$								a_i
		1	2	3	4	5	6	7	8	
1	$\langle \emptyset, \{1,3,5\}, \{1,6,8\}, \{1,3,6\}, \{1,6,8\}, \{6,7,8\} \rangle$	4	$\underline{0}$	2	0	1	4	1	3	2
2	$\langle \{2\}, \{1,6,8\}, \{1,3,6\}, \{1,6,8\}, \{6,7,8\} \rangle$	3	1	1	$\underline{0}$	0	4	1	3	4
3	$\langle \{2,4\}, \{1,3,6\}, \{1,6,8\}, \{6,7,8\} \rangle$	2	1	1	1	$\underline{0}$	3	1	2	5
4	$\langle \{2,4,5\}, \{1,6,8\}, \{6,7,8\} \rangle$	1	1	$\underline{0}$	1	1	2	1	2	3
5	$\langle \{2,4,5,3\}, \{6,7,8\} \rangle$	$\underline{0}$	1	1	1	1	1	1	1	1

Also here an (L, U)-bounded priority queue, where $L = 1$ and $U = n$, can be used to store the vertices with frequency 0, providing a fast way of identifying the least numbered one. The algorithm is very similar to the encoding one, having also worst-case time complexity of $O(m)$.

In the following algorithm, a valid Rényi code $B = \langle B_1, \dots, B_{n-k-1} \rangle$ is given as input, along with the value of n, and the elements $a_1, \dots, a_{n-k}$ are obtained as output.

```
procedure Decode;
begin
    k ← n − |B| − 1;
    B_{n−k} ← {n − k + 1, ..., n};
    MakeEmptyQueue(Q, 1, n);
    for i ← 1, ..., n do
        freq(i) ← how many times i appears in B;
        if freq(i) = 0 then
            Insert(Q, i, i);
    for i ← 1, ..., n − k − 1 do
        DeleteMin(Q, a_i, a_i);
        freq(a_i) ← 1;
        for j ∈ B_i do
            freq(j) ← freq(j) − 1;
            if freq(j) = 0 then
                Insert(Q, j, j);
    DeleteMin(Q, a_{n−k}, a_{n−k});
end.
```

6 Conclusions

The (L, U)-bounded priority queue, introduced in this paper, particularizes ordinary priority queues in two aspects: the priorities associated to the elements must be integer numbers constrained to a predefined interval and, in a sequence of operations, no more than one $Insert$ can immediatly follow a $DeleteMin$. We demonstrated through an amortized analysis that, in the worst case, a valid sequence of operations containing exactly t insertions can be executed in time $O(t + \Delta)$, where Δ is the amplitude of the range of priorities.

The first of these restrictions occurs very often in practical applications, where the range of priorities is normally known *a priori*. By forbidding the occurence of more than one $Insert$ after a $DeleteMin$ operation, we are actually addressing a class of scheduling problems in which the precedence digraph of the involved tasks has an *in-tree* structure, i.e., every task is a prerequisite for at most another task. Thus, supposing that the priority queue contains the tasks which are *ready* to be executed, the completion of one task must introduce at most one new task into the set of ready-to-run ones.

Whenever these two assumptions hold, (L, U)-bounded priority queues can be applied as a fast way of identifying the element with the best priority in a set.

References

1. Blair, J. R. S., Peyton, B., An Introduction to Chordal Graphs and Clique Trees, In: *Graph Theory and Sparse Matrix Computation*, IMA 56, pp. 1–29, 1993.
2. Caminiti, S., Finocchi, I., Petreschi, R., A Unified Approach to Coding Labelled Trees, *Proceedings of the 6^{th} LATIN'04*, pp. 339–348, 2004.
3. Chen, W., A Coding Algorithm for Rényi Trees, *Journal of Combinatorial Theory*, series A, vol. 63, pp. 11–25, 1993.
4. Chen, H.-C., Wang, Y.-L., An Efficient Algorithm for Generating Prüfer Codes from Labelled Trees, *Theory of Computing Systems*, vol. 33, pp. 97–105, 2000.
5. Deo, N., and P. Micikevicius, Prüfer-like Codes for Labeled Trees, *Congressus Numerantium*, vol. 151, pp. 65–73, 2001.
6. Golumbic, M. C., *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.
7. Harary, F., and E. M. Palmer, On Acyclic Complexes, *Mathematika*, vol. 15, pp. 115–122, 1968.
8. Justel C. M., and L. Markenzon, Lexicographic Breadth First Search and k-Trees, *Proceedings of JIM'2000* - Secondes Journées de l'Informatique Messine, pp. 23–28, Metz, France, 2000.
9. Prüfer, A., Neuer Beweis eines Satzes über Permutationen, *Archiv der Mathematik und Physik*, vol. 27, pp. 142–144, 1918.
10. Pereira, P. R. C., Markenzon, L., Vernet, O., The Reduced Prüfer Code for Rooted Labelled k-Trees, *Electronic Notes in Discrete Mathematics*, vol 22, pp. 135–139, 2005.
11. Rose, D. J., On Simple Characterizations of k-Trees, *Discrete Mathematics*, vol. 7, pp. 317–322, 1974.

12. Rényi, C., and A. Rényi, Prüfer Code for k-Trees, In: P. Erdös *et al.*, editor, *Combinatorial Theory and its Applications*, pp. 945–971, 1970.
13. Weiss, M. A., *Data Structures and Algorithm Analysis*, Addison-Wesley, 1995.