Coloring Permutation Graphs in Parallel

Stavros D. Nikolopoulos

5-99

Technical Report No. 5-99/1999

Department of Computer Science
University of Ioannina
45 110 Ioannina, Greece

Coloring Permutation Graphs in Parallel

Stavros D. Nikolopoulos

Department of Computer Science, University of Ioannina,
P.O. Box 1186, GR-45110 Ioannina, Greece.
e-mail: stavros@cs.uoi.gr

Abstract — We study the problem of coloring permutations graphs using some properties of the Lattice representation of a permutation and the relationship between permutations and binary search trees. We propose an efficient parallel algorithm which colors a permutation graph in $O(\log^2 n)$ time using $O(n^2 / \log n)$ processors on the CREW PRAM model, where n is the number of vertices in the permutation graph. Specifically, given a permutation π we construct a tree $T^*[\pi]$, which we call *coloring-permutation tree*, using certain combinatorial properties of π. We show that the problem of coloring a permutation graph is equivalent to finding vertex levels in the color-permutation tree. Our results improve in performance upon the best-known parallel algorithms for the same problem.

Keywords: Permutation graphs, Perfect graphs, Coloring problem, Parallel algorithms, Trees, Complexity, PRAM models.

1. Introduction

Let $N = \{1, 2, ..., n\}$ and $\pi = [\pi_1, \pi_2, ..., \pi_n]$ be a permutation on N. Then we construct an undirected graph $G[\pi]$ in the following manner: $G[\pi] = (V, E)$ has vertices numbered from 1 to n, that is $V = \{1, 2, ..., n\}$, and

$$(i, j) \in E \iff (i - j)\,(\pi_i^{-1} - \pi_j^{-1}) < 0$$

for all $i, j \in N$, where π_i^{-1} denotes the position of number i in π [5, 13]. An undirected graph G is called *permutation graph* if there exist a permutation π such that G is isomorphic to $G[\pi]$. We, therefore, assume in this paper that a permutation graph $G[\pi]$ is represented by the corresponding permutation π. The graph $G[\pi]$ is sometimes called the *inversion graph* of π [5].

Many researchers have been devoted to the study of permutation graphs. They have proposed sequential and/or parallel algorithms for recognizing permutations graphs and solving combinatorial and optimization problems on them. For a sequential environment, Pnueli *et. al.* [11] gave an $O(n^3)$ time algorithm for recognizing permutation graphs using the transitive orientable graph test. Later, Spinrad [14] improved their results by designing an $O(n^2)$ time algorithm for the same problem. In the same paper, Spinrad also proposed an algorithm that determines whether or not two permutation graphs are isomorphic in $O(n^2)$ time. In [15], Spinrad *et. al.* proved that a bipartite permutation graph can be recognized in linear time by using some good algorithmic properties of such a graph. They also studied other combinatorial and optimization problems on permutation graphs. Supowit [16] solved the coloring problem, the maximum clique problem, the cliques cover problem and the maximum independent set problem, all in $O(n\log n)$ sequential time. Moreover, Farber and Keil [4] solved the weighted domination problem and the weighted independent domination problem in $O(n^3)$ time, using dynamic programming techniques. Later, Brandstadt and Kratsch [3] published an $O(n^2)$ time algorithm for the weighted independent domination problem. Atallah *et. al.* [1] solved the independent domination set problem in $O(n \log^2 n)$ time, while Tsai and Hsu [17] solved the domination problem and the weighted domination problem in $O(n\log^2 n)$ time and $O(n^2\log^2 n)$ time, respectively. Tsukiyama *et. al.* [18] proposed an algorithm that generates all maximal independent sets of a general graph in $O(nma)$ time, where a is the number of the generated maximal independent sets of the graph. Leung [10] gave algorithms for generating all maximal independent sets of interval, circular-arc and triangulated (or chordal) graphs. His algorithm takes $O(n^2+k)$ time for interval and circular-arc graphs, and $O((n+m)a)$ time for triangulated graphs, where k is the number of vertices generated. In [20], Yu and Chen showed that the generation of all the maximal independent sets can be completed in $O(n\log n+k)$ time using $O(n\log n)$ space.

Although many sequential algorithms have been proposed for permutations graphs, few parallel algorithm have been appeared in the literature. Due to work of Helmbold and Mayr [6] and Kozen *et. al.* [9], the problem of recognizing permutation graphs was shown to be in the NC class. Helmbold and Mayr presented a parallel algorithm that recognizes a permutation graph in $O(\log^3 n)$ time using $O(n^4)$ processors on a CRCW PRAM model. They also solved the weighted clique problem and the coloring problem in $O(\log^3 n)$ time using $O(n^4)$ processors on same model of computation. Moreover, given a permutation graph, their algorithm can construct the permutation that represents the permutation graph.

Our objective is to study the coloring problem of permutation graphs. Recently, Yu and Chen [19] proposed a technique that transfer the coloring problem into the largest-weight path problem. Specifically, their technique, first, transforms a permutation graph (combinatorial object) into a set of planar points (geometrical object), then constructs an acyclic directed graph by exploiting the domination relation within the geometric object and, finally, solves the largest-weight path problem on the acyclic directed graph. The parallel algorithm they proposed can solve the coloring problem in $O(\log^2 n)$ time with $O(n^3/\log n)$ processors on a CREW PRAM, or in $O(\log n)$ time with $O(n^3)$ processors on a CRCW PRAM model. Moreover, they proposed parallel algorithms that solve the weighted clique problem, the weighted independent set problem, the cliques cover problem, and the maximal layers problem with the same time and processor complexities.

In this paper, we present a parallel algorithm for the problem of coloring a permutation graph, which run in $O(\log^2 n)$ time with $O(n^2/\log n)$ processors on the CREW PRAM model. Our algorithm is based on some properties of the Lattice representation of a permutation and the relationship between permutations and binary search trees. Specifically, given a permutation π, we construct a tree $T^*[\pi]$ using certain combinatorial properties of π. We call this tree *coloring-permutation tree* (cp-tree) or *color tree* for short. We prove that the problem of coloring a permutation graph $G[\pi]$ is equivalent to the problem of finding the level of each node of the color tree $T^*[\pi]$. We show that the color tree of a permutation can be constructed in $O(\log^2 n)$ time with $O(n^2/\log n)$ processors on the CREW PRAM model. Since, the level of each vertex of a tree is computed in $O(\log n)$ time with $O(n/\log n)$ processors on the EREW PRAM model using the well-known Euler tour technique, it follows that the coloring problem on permutation graphs can be solved in $O(\log^2 n)$ time with $O(n^2/\log n)$ processors on the CREW PRAM model.

The paper is organised as follows. In Section 2, we establish the notation and terminology we shall use throughout the paper. In Section 3, we describe the method that transforms a given permutation π into a rooted tree, that is, the color tree, and we prove that coloring the permutation graph $G[\pi]$ is equivalent to the problem of finding the level of each node of its color tree. In Section 4, we present a parallel algorithm for the construction of the color tree, while in Section 5 we prove the correctness of the construction algorithm. In Section 6, we compute the time and processor complexity of the coloring algorithm. Finally, Section 5 concludes the paper.

2. Definitions

A *coloring* of a graph $G = (V, E)$ is an assignment of colors to its vertices so that no two adjacent vertices have the same color. The set of all vertices with any one color is independent and is called a *color class*. To distinguish the color classes we use a set of colors C, and the division into color classes is given by a *coloring* $\varphi: V \to C$, where $\varphi(x) \neq \varphi(y)$ for all $(x, y) \in E$. If C has cardinality k, then φ is a *k-coloring*. The *coloring problem* is to color a graph G with k color where k is the minimum cardinal k for which G has a k-coloring (minimum number of colors). The number k is called chromatic number of G and denoted by $\chi(G)$ [6, 8].

We have seen that an undirected graph G is a permutation graph if there exist a permutation π such that G is isomorphic to $G[\pi]$ (see Introduction). Let us now give some basic properties of permutations. It is well-known that permutations may be represented in many ways. The most straightforward way to represent permutations is simply as a rearrangement of the numbers 1 through n; hereafter $N_n = \{1, 2, ..., n\}$. For example, in the following permutation $n = 9$.

index	1	2	3	4	5	6	7	8	9
permutation	7	9	3	6	2	8	5	4	1

Suppose π is a permutation on N_n. Let us think of π as the sequence $[\pi_1, \pi_2, ..., \pi_n]$, so, for example, the permutation $\pi = [7, 9, 3, 6, 2, 8, 5, 4, 1]$ has $\pi_1 = 7$, $\pi_2 = 9$, etc. Notice that $(\pi^{-1})_i$, denoted here as π_i^{-1}, is the position of element i in the sequence π. In our example $\pi_7^{-1} = 1$, $\pi_9^{-1} = 2$, etc.

Let $\pi = [\pi_1, \pi_2, ..., \pi_n]$ be a permutation on N_n. An *inversion* is a pair $i < j$ with $\pi_i > \pi_j$. We say that an element π_i *inverts* π_j, or π_j is *inverted* by π_i, if $i < j$ and $\pi_i > \pi_j$. An element π_i *directly inverts* π_j, or π_j is *directly inverted* by π_i, if π_i *inverts* π_j and there exists no element π_k such that π_i inverts π_k and π_k inverts π_j. In the permutation given above, the elements 2, 5, 4 and 1 are inverted by 6, while the elements 2 and 5 are directly inverted by 6. We shall use, hereafter, the notation *D-inverts* and *D-inverted* for the terms directly inverts and directly inverted, respectively.

If q_j is the number of $i < j$ with $\pi_i > \pi_j$, then the sequence $[q_1, q_2, ... q_n]$ is called the *inversion table* of π. In other words, q_j is the number of elements that invert π_j. Similarly, if p_j is the number of elements that D-invert π_j, then the sequence $[p_1, p_2, ... p_n]$ is called the *D-inversion table* of π. The sample permutation given above has the following inversion and D-inversion tables: *inversion-table*$(\pi) = [0, 0, 2, 2, 4, 1, 4, 5, 8]$ and *D-inversion-table*$(\pi) = [0, 0, 2, 2, 2, 1, 2, 1, 2]$. We can see that the zero entries in the inversion or D-inversion table corresponds to those elements of π that are not inverted by any other element in π.

The *inversion set* (resp. *D-inversion set*) of an element π_i is defined to be the set which contains all the elements of π that invert (resp. D-invert) π_i. We shall denote the inversion and D-inversion sets of an element π_i by *inversion-set*(π_i) and *D-inversion-set*(π_i), respectively. In our example, these two sets of the element $\pi_7 = 5$ are the following: *inversion-set*(5) = {7, 9, 6, 8} and *D-inversion-set*(5) = {6, 8}.

Fig. 1 shows a two-dimensional representation of a permutation that is useful for showing the inversion and D-inversion sets of its elements. The permutation $[\pi_1, \pi_2, ..., \pi_n]$ is represented by labelling the cell at row i and column π_i with the element π_i for each i. There is one label in each row and in each column, so each cell in the lattice corresponds to a unique pair of labels. If one member of the pair is below and the other to the right, then that pair is an inversion in the permutation. Based on this property we can easily show the inversion and D-inversion relations of every pair of elements. For example, let π_i, π_j be a pair of element of the permutation π. If π_i is below and π_j to the right, then π_j inverts π_i (or π_i is inverted by π_j). In Fig. 1 (leftmost lattice), this relation is indicated by a bullet in the corresponding cell, that is, in row j and column π_i. The D-inversion relation of each pair of elements of π is indicated in a similar way in the rightmost lattice of Fig. 1.

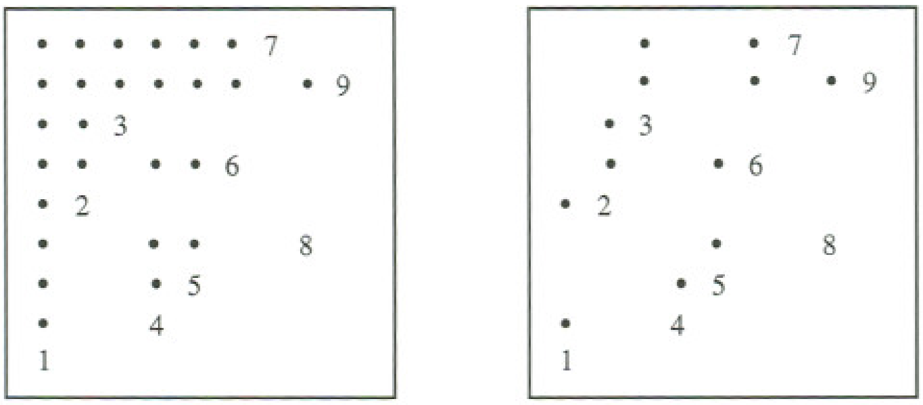

Fig. 1: Lattice representation of the permutation $\pi = [7, 9, 3, 6, 2, 8, 5, 4, 1]$ and its D-inversion relation of each pair of elements of π.

Given a permutation π on N_n, its corresponding permutation graph $G[\pi] = (V, E)$ can be constructed as follows: $G[\pi]$ has vertices numbered from 1 to n; two vertices are jointed by an edge if the larger of their corresponding numbers is to the left of the smaller in permutation π. That is, $G[\pi]$ has n vertices 1, 2, ..., n, and m edges such that $(i, j) \in E$ if and only if $(i, j)\ (\pi_i^{-1} - \pi_j^{-1}) < 0$.
conclude this section with some graph-theoretic notation employed in this paper. A *tree* T = (V, E) is a graph with a unique path between each pair of vertices. A *rooted tree* has a distinguished vertex called the root. A *directed tree* is a rooted tree with directed edges. A (*directed*) *forest* is a collection of (directed) trees. Throughout the paper, all trees will be directed.

We shall consider the directed trees to be *leveled*; that is, the root r will constitute level 0, the neighbours of r will constitute level 1, the neighbours of the vertices on level 1 that have not yet placed in a level will constitute level 2, etc. It is well-known that with this structure, if u is on level h

then the children of u are on level h+1 and the parent of u is on level h-1. Throughout the text, we shall refer to the level of u as *level*(u). It is easy to see that *level*(u) is simply the *length* of the path (number of edges) from the root r to u or, equivalently, the *distance* (number of edges) between u and the root r. The *height* of a node u is the number of edges in the longest path from the node u to a leaf. Finally, we define the *height of a tree* to be the height of its root.

3. The Color-Mapping Strategy

We have referred to the problem of coloring a graph as one of trying to assign particular colors to its vertices so that no two adjacent vertices have the same color. Moreover, the number of colors used must be as less as possible. The key to the solution is to find the color classes of the graph; that is, the classes of vertices that can be colored with the same color. To this end, one can think of transforming the graph into another combinatorial object (e.g., tree, directed graph, etc.) and, then, solving a particular problem on this object (e.g., vertices lying in the same level of the tree, vertices having the same distance from a particular vertex, etc.) which gives the solution to the coloring problem.

In this work, we use a strategy to transform a permutation graph $G[\pi]$ into a rooted tree $T^*[\pi]$, which we call *coloring-permutation tree* or *cp-tree* for short. Then, we solve the coloring problem on $G[\pi]$ by computing the vertex-level function on $T^*[\pi]$. More precisely, given a permutation π (or its corresponding graph), we construct a coloring-permutation tree $T^*[\pi]$ by exploiting the inversion and D-inversion relation and we show that the color class C_i of graph $G[\pi]$ consists of those nodes of the tree $T^*[\pi]$ whose distance from the root of the tree equals $i \geq 1$. That is, C_i contains all the nodes u of $T^*[\pi]$ such that $level(u) = i$, $1 \leq i \leq k$, where $k = \chi(G[\pi])$.

Towards the construction of a coloring-permutation tree $T^*[\pi]$, we first construct a rooted tree $T[\pi]$ as follows:

(i) Construct a directed acyclic graph (dag) $G = (V, E)$ such that $V = \{\pi_1, \pi_2, ..., \pi_n\}$ and $<\pi_i, \pi_j> \in E$ iff π_i D-inverts π_j; (see Fig. 2: leftmost figure).

(ii) Given the dag G, construct a (directed) forest $\mathcal{F}$ as follows: Remove the edge $<\pi_j, \pi_k>$ from G iff there exists edge $<\pi_i, \pi_k>$ such that $\pi_i < \pi_j$. The node π_i with indegree(π_i) = 0 in G is the root of the tree T_i in $\mathcal{F}$; (see Fig. 2: middle figure).

(iii) Let $T_1, T_2, ..., T_k$ ($k \geq 1$) be the trees in $\mathcal{F}$, and let $\pi_1, \pi_2, ..., \pi_k$ be the roots of those trees, respectively. Let r be a new node such that $r > \pi_i$ for every i, $1 \leq i \leq n$. Then, construct a rooted tree $T[\pi]$ consisting of the nodes and edges of $T_1, T_2, ..., T_k$, the new node r, and new edges $<r, \pi_1>, <r, \pi_2>, ..., <r, \pi_k>$. The root of $T[\pi]$ is r, and $T_1, T_2, ..., T_k$ are the subtrees of $T[\pi]$; (see Fig. 2; rightmost figure).

The tree $T[\pi]$ constructed by the above procedure has the property that every path from the root r to a node π_i forms a decreasing sequence $P = [r, \pi_p, ..., \pi_q]$. Moreover, if π_i and π_j are two elements of P such that $\pi_i > \pi_j$ ($\pi_i \neq r$ and $\pi_j \neq r$), then $\pi_i^{-1} < \pi_j^{-1}$ in π. Based in this property we shall refer, hereafter, to the tree $T[\pi]$ as *decreasing-subsequence tree*, or *ds-tree* for short.

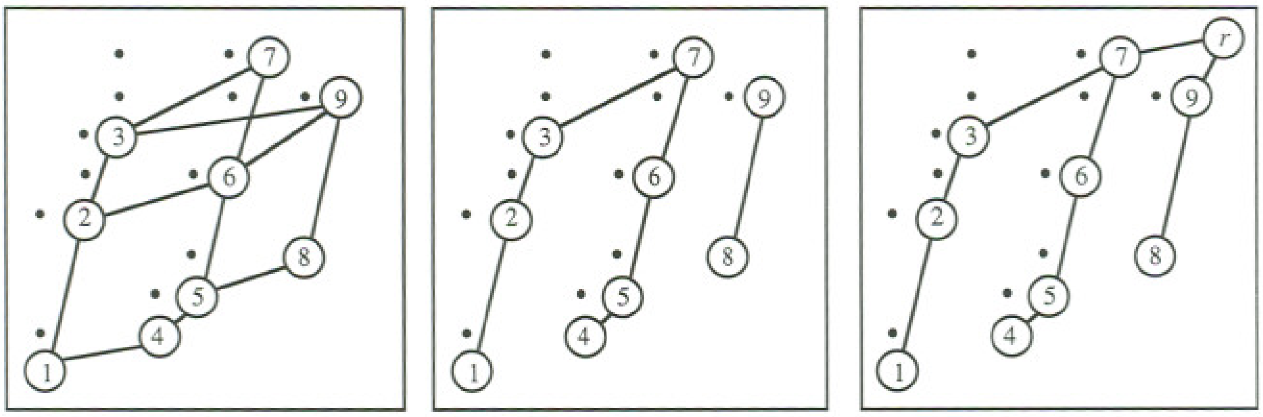

Fig. 2: The construction of the tree T[π] of the permutation $\pi = [7, 9, 3, 6, 2, 8, 5, 4, 1]$.

Let $\pi = [\pi_1, \pi_2, ..., \pi_n]$ be a permutation on N_n. We define a coloring-permutation tree $T^*[\pi] = (V^*, E^*)$ to be a rooted tree having the following properties:

(i) $V^* = \{r, \pi_1, ..., \pi_n\}$, where r is the root of the tree, and $r > \pi_i$ for every i, $1 \leq i \leq n$.

(ii) $<\pi_i, \pi_j> \in E^*$ if π_i inverts π_j in π.

(iii) there is no pair of nodes π_i, π_j such that $level(\pi_i) \geq level(\pi_j)$ and π_i inverts π_j in π.

It is easy to see that the ds-tree T[π] of a permutation π is a cp-tree $T^*[\pi]$ if there are no nodes π_i, π_j in T[π] such that $level(\pi_i) \geq level(\pi_j)$ and π_i inverts π_j in π. Fig. 3 shows the ds-tree T[π] and a cp-tree $T^*[\pi]$ of the permutation $\pi = [7, 9, 3, 6, 2, 8, 5, 4, 1]$. In this figure, T[$\pi$] is not a cp-tree because there is a pair of nodes {4, 1} such that $level(4) \geq level(1)$ and 4 inverts 1.

Having constructed the ds-tree T[π] of a permutation π, let us now show the way we can construct the cp-tree $T^*[\pi]$. Let π_i, π_j be two nodes of the ds-tree T[π] such that $level(\pi_i) \geq level(\pi_j)$ and π_i inverts π_j in π. Then, we define the following operation on the ds-tree T[π]:

- **Inversion-removing**: Let $p(\pi_j)$ be the parent of the node π_j in T[π]. Delete the edge $<p(\pi_j), \pi_j>$ from and add a new edge $<\pi_i, \pi_j>$ to T[π].

Let T'[π] be the resulting tree after applying an Inversion-removing operation on T[π]. Since π_i inverts π_j, it follows that $\pi_i > \pi_j$ and $\pi_i^{-1} < \pi_j^{-1}$ in π. Therefore, every path from the root of T'[π] to a node π_i forms a decreasing sequence of π. Thus, we can construct a cp-tree $T^*[\pi]$ by applying Inversion-removing operations on the ds-tree T[π] until no pair of nodes $\{\pi_i, \pi_j\}$ remains in T[π] such that $level(\pi_i) \geq level(\pi_j)$ and π_i inverts π_j in π. For example, let T[π] be the ds-tree of the sample permutation π used throughout the paper (see Fig. 3; leftmost tree). In this tree, $level(4) = level(1)$ and 4 inverts 1 in π. It is easy to see that we can construct a cp-tree $T^*[\pi]$ by applying an Inversion-removing operation on the ds-tree T[π] (see Fig. 3; rightmost tree).

Remark 3.1 Let $G[\pi]$ be a permutation graph and let $T^*[\pi]$ be a ds-tree of π rooted at r. Based on the way we construct the graph $G[\pi]$ from the permutation π and the way we construct the cp-tree $T^*[\pi]$ from the ds-tree T[π], we conclude that if $P = [r, \pi_i, ..., \pi_j]$ is a path in $T^*[\pi]$, then the subgraph of $G[\pi]$ induced by $\{\pi_i, ..., \pi_j\}$ is a K_m, where m is the number of elements in P.

We now show that there is an one-to-one correspondence between the length (number of edges) in the path from r to a node π_i in $T^*[\pi]$ and the color of vertex π_i in $G[\pi]$. More precisely, we prove that the nodes of the ith level of the cp-tree $T^*[\pi]$ form the color class C_i of the permutation graph $G[\pi]$, where $1 \le i \le k$. Recall that $k = \chi(G[\pi])$.

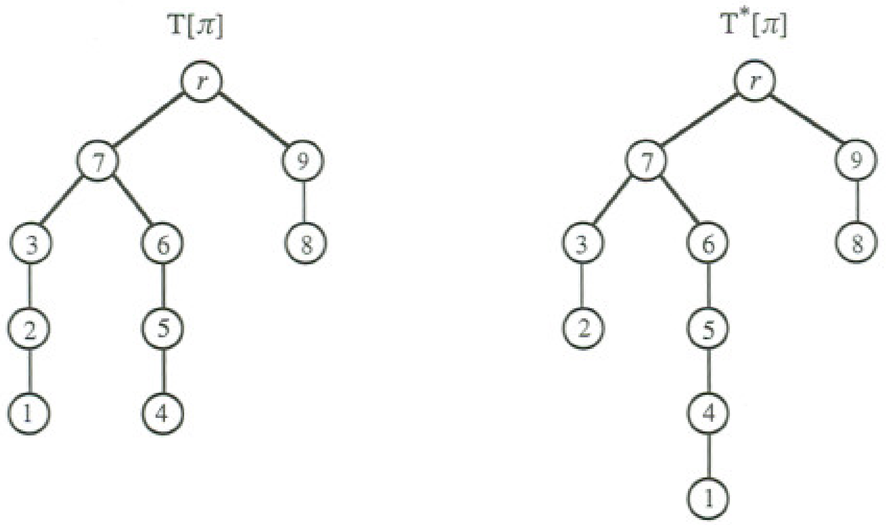

Fig. 3: The decreasing-subsequence tree $T[\pi]$ and a coloring-permutation tree $T^*[\pi]$ of the permutation $\pi = [7, 9, 3, 6, 2, 8, 5, 4, 1]$.

Lemma 3.1 Let π be a permutation on N_n. The following numbers are equal:

(i) the chromatic number of $G[\pi]$;

(ii) the length of a longest decreasing subsequence of π;

Proof. The equivalence of (i) and (ii) holds since a longest subsequence of π corresponds to a maximal clique of $G[\pi]$, which will be of size $\chi(G[\pi])$ since permutation graphs are perfect (see also Corollary 7.4 in [5]). □

Lemma 3.2 Let $T^*[\pi]$ be a cp-tree of a permutation π rooted at r. Every path from r to a node π_i forms a decreasing subsequence of π.

Proof. The lemma holds since the Inversion-moving operation has respect for the properties of the ds-tree $T[\pi]$. □

Lemma 3.3 Let $T^*[\pi]$ be a cp-tree of a permutation π rooted at r and let k be the height of $T^*[\pi]$. Then $k = \chi(G[\pi])$.

Proof. Lemma 3.2 tell us that every path from r to a node π_i of $T^*[\pi]$ forms a decreasing subsequence of π. This result coupled with the result of Lemma 3.1 implies that $k \le \chi(G[\pi])$ (Note that the length of a subsequence S of π is the number of elements in S, while the length of a path P of $T^*[\pi]$ is the number of edges in P.) Suppose that $k < \chi(G[\pi])$. Let $S = [\pi_p, ..., \pi_q]$ be the longest decreasing subsequence of π. Since the length of S equals $\chi(G[\pi])$ and $k < \chi(G[\pi])$, it follows that there are π_i and π_j in S such that $\pi_i > \pi_j$ and $level(\pi_i) \ge level(\pi_j)$ in $T^*[\pi]$. Moreover, since S is a subsequence of π, it follows that $\pi_i^{-1} < \pi_j^{-1}$ in π. Thus, $T^*[\pi]$ is not a cp-tree; a contradiction. □

Lemma 3.4 Let π_i and π_j be nodes in $T^*[\pi]$. If (π_i, π_j) is an edge in $G[\pi]$, then $level(\pi_i) \neq level(\pi_j)$.

Proof. Suppose that $level(\pi_i) = level(\pi_j)$. Since (π_i, π_j) is an edge in $G[\pi]$, it follows that π_i inverts π_j in π. Thus, there is pair of nodes $\{\pi_i, \pi_j\}$ in $T^*[\pi]$ such that $level(\pi_i) \geq level(\pi_j)$ and π_i inverts π_j in π, contradicting the properties of a cp-tree. □

Having shown the relation between the coloring problem on a permutation graph $G[\pi]$ and the problem of finding the level of each node of the cp-tree $T^*[\pi]$, we are in a position to formulate an algorithm for solving the coloring problem on permutation graphs. The algorithm proceeds as follows:

```
Algorithm Coloring:
Input  : A permutation $\pi$ and its corresponding graph $G[\pi]=(V, E)$;
Output : The color of each vertex $\pi_i \in V$, $i=1, 2, ..., n$;
begin
1. Construct a coloring-permutation tree $T^*[\pi]$ rooted at $r$, where $r = \pi_0$;
2. Compute the level $level(\pi_i)$ of each node $\pi_i$ of the tree $T^*[\pi]$, $1 \leq i \leq n$;
3. Set $color(\pi_i) \leftarrow level(\pi_i)$, $i = 1, 2, ..., k$;
end;
```

In step 3, the algorithm colors the vertices of graph $G[\pi]$ with k colors, where k is the height of the color tree $T^*[\pi]$, $k \leq n$. Vertices π_i and π_j are colored with the same color if and only if the nodes π_i and π_j have the same distance in $T^*[\pi]$ or, equivalently, $level(\pi_i) = level(\pi_j)$. The correctness of the algorithm is established through the Theorem 3.1. Its proof relies on the results of Lemmas 3.3 and 3.4. Hence we may state the following:

Theorem 3.1 Given a permutation π, the algorithm `Coloring` correctly solves the coloring problem on the permutation graph $G[\pi]$.

4. Construction of the Coloring-Permutation Tree

We have defined the coloring-permutation tree $T^*[\pi]$ of a given permutation π and we have showed the one-to-one correspondence between the coloring problem on $G[\pi]$ and the problem of computing the level of each vertex of $T^*[\pi]$. It is well-known that we can optimally compute the level of each vertex of $T^*[\pi]$ using the Euler-tour technique on rooted trees. Thus, we focus on the design of a parallel algorithm for constructing a coloring-permutation tree $T^*[\pi]$.

4.1 The decreasing-subsequence trees

As we showed in the previous section, the ds-tree $T[\pi]$ is constructed by exploiting the *D-inversion* relation on the permutation π. Therefore, there is a need of computing the *D*-inversion set for every element of π. Obviously, the *D*-inversion set of an element is subset of its inversion set. So, we first compute the inversion set for every element π_i in π, and then select from it the elements that constitute the *D*-inversion set.

We can easily see that, the inversion set of an element π_i of a permutation π is simply the set which contains all the elements that are greater than π_i and lie on the left of the element π_i in π (see the definition of the inversion set in Section 2). That is,

$$\textit{inversion-set}(\pi_i) = \{\pi_j \mid \pi_j > \pi_i \text{ and } j < i\}$$

Let $B_i\,[1..i\text{-}1]$ be an array such that $B_i[j] = \pi_j$ if $\pi_j \in \textit{inversion-set}(\pi_i)$; otherwise $D_i[j] = \infty$ for $j = 1, 2, ..., i\text{-}1$. By definition, an element π_i is *D*-inverted by π_j (π_j *D*-inverts π_i) if $\pi_j \in \textit{inversion-set}(\pi_i)$ and there is no element π_k such that π_j inverts π_k and π_k inverts π_i. Thus, the last element π_j of *inversion-set*(π_i) is a member of *D-inversion-set*(π_i). That is, π_j is the element of *inversion-set*(π_i) with the *max* index in π. It is easy to see that $\pi_k \in \textit{D-inversion-set}(\pi_i)$ if $\pi_k \in \textit{inversion-set}(\pi_i)$ and $\pi_k < \pi_{k'}$ for every $\pi_{k'}$ in $B_i[k..i\text{-}1] = [\pi_k, \pi_{k+1}, ..., \pi_{i-1}]$. Therefore, we can compute the *D-inversion-set* of an element π_i by computing the suffix minima of $B_i[1..i\text{-}1] = [\pi_1, ..., \pi_k, ..., \pi_{i-1}]$, where $\pi_k = \infty$ if $\pi_k \notin \textit{inversion-set}(\pi_i)$, $1 \le k \le i\text{-}1$.

index	1	2	3	4	5	6	7	8	9
permutation	7	9	3	6	2	8	5	4	1

7	9	3	6	2	8	**5**	4	1

7	9	∞	6	∞	8	**5**	0	0

6	6	6	6	8	8	**5**	0	0

(a)

7	0	0	0	0	0	0	0	0
∞	**9**	0	0	0	0	0	0	0
7	9	**3**	0	0	0	0	0	0
7	9	∞	**6**	0	0	0	0	0
3	3	3	6	**2**	0	0	0	0
9	9	∞	∞	∞	**8**	0	0	0
6	6	6	6	8	8	**5**	0	0
5	5	5	5	5	5	5	**4**	0
2	2	2	2	2	4	4	4	**1**

(b)

Fig. 4: **(a)** The computation of the 7th row of the *D*-inversion matrix of $\pi = [7, 9, 3, 6, 2, 8, 5, 4, 1]$; **(b)** The *D*-inversion matrix *D* of π.

Let $D_i[1..i\text{-}1]$ be an array containing that suffix minima of $B_i[1..i\text{-}1]$, $2 \le i \le n$. Obviously, $D_i[1]$ is the minimum element among all the elements π_j of π such that $j < i$ and $\pi_j > \pi_i$. That is, $D_i[1] = \min\{\pi_j \mid \pi_j \in \textit{inversion-set}(\pi_i)\}$. From the way we construct the ds-tree T[π], we can easily see that if $p(\pi_i)$ is the parent of node π_i in T[π], then $p(\pi_i) = D_i[1]$.

In Fig. 4(a), we show the permutation $\pi = [7, 9, 3, 6, 2, 8, 5, 4, 1]$, the array $B_7[1..6]$ and the array $D_7[1..6]$. We observe that $p(\pi_7) = 6$, where $\pi_7 = 5$ (see also Fig. 3).

Having computed the *D-inversion-set* of an element π_i of π; that is, the array $D_i[1..i\text{-}1]$, let us now define an $n \times n$ matrix D which contains all the necessary information for the D-inversion relation of the permutation π. We call this matrix *D-inversion matrix* and we define it as follows:

(i) $D[i, i] = \pi_i$, for $i = 1, 2. ..., n$;
(ii) $D[i, p] = 0$ for $i = 1, 2. ..., n$; and $p = i+1, i+2. ..., n$;
(iii) $D[1..i\text{-}1] = D_i[1..i\text{-}1]$ for $i = 1, 2. ..., n$;

Thus, we can easily see that Fig. 4(a) shows the computation of the 7th row of the D-inversion matrix D of the permutation $\pi = [7, 9, 3, 6, 2, 8, 5, 4, 1]$. Fig. 4(b) shows the D-inversion matrix D of the same permutation π.

Let E_π be the edge set of the ds-tree $T[\pi]$ rooted at r. Since r inverts every element of π, it follows that $<r, D[1, 1]> \in E_\pi$. From the D-inversion matrix D we obtain that $<r, D[1, i]> \in E_\pi$ if $D[1, i] = \infty$ or $<D[i, i], D[1, i]> \in E_\pi$ if $D[1, i] \neq \infty$, $2 \leq i \leq n$. Thus, the D-inversion matrix contains all the necessary information for the construction of the ds-tree $T[\pi]$.

Next we give a more formal listing of the algorithm for computing the D-inversion matrix of a permutation π. The computation of each D_i, $1 \leq i \leq n$, can be done independently, and therefore in parallel. The following parallel algorithm describes this computation.

```
Algorithm D-inversion-matrix:
Input  : A permutation $\pi$ on $\{1, 2, ..., n\}$;
Output : The D-inversion matrix $D$ of the permutation $\pi$;
begin
1. For every $i$, $1 \leq i \leq n$, do in parallel
   1.1 $D[i, i] \leftarrow \pi_j$;
   1.2 $D[i, j] \leftarrow 0$ for $i+1 \leq j \leq n$;
   1.3 If $D[i, j] < D[i, i]$ then $D[i, j] \leftarrow \infty$, for $1 \leq j \leq i\text{-}1$;
   1.4 Compute the suffix minima of $D[i, 1..i\text{-}1]$;
end;
```

The correctness of the algorithm is based on the previous discussion and is established through the following Lemma.

Lemma 4.1 Algorithm `D-inversion-matrix` correctly computes the D-inversion matrix of a permutation π on N_n.

Let $\pi = [r, \pi_1, \pi_2, ..., \pi_n]$ be a permutation such that $r > \pi_i$ for every i, $1 \leq i \leq n$. We have seen that the ds-tree $T[\pi]$ is a rooted tree such that: r is the root of the tree; π_i is a node iff $r > \pi_i$; $<\pi_k, \pi_i>$ is an edge iff $\pi_k = min\{\pi_j \mid \pi_j \in D\text{-}inversion\text{-}set(\pi_i)\}$; $1 \leq i \leq n$.

Next, we define the ds-trees $T[\pi_1]$, $T[\pi_2]$, ..., $T[\pi_n]$ of a permutation $\pi = [\pi_1, \pi_2, ..., \pi_n]$. The ds-tree $T[\pi_i]$ $(1 \leq i \leq n)$ is defined to be a rooted tree such that: π_i is the root of the tree; π_p is a node iff $\pi_i > \pi_p$ and π_p in $[\pi_{i+1}, \pi_{i+2}, ..., \pi_n]$; $<\pi_k, \pi_p>$ is an edge iff $\pi_k = min\{\pi_j \mid \pi_j \in D\text{-}$

inversion-set(π_p)}; $i+1 \leq p \leq n$. Fig. 5 illustrates the way we can construct the ds-trees T[7], T[9] and T[3] of the permutation π, where π is sample permutation used in this paper.

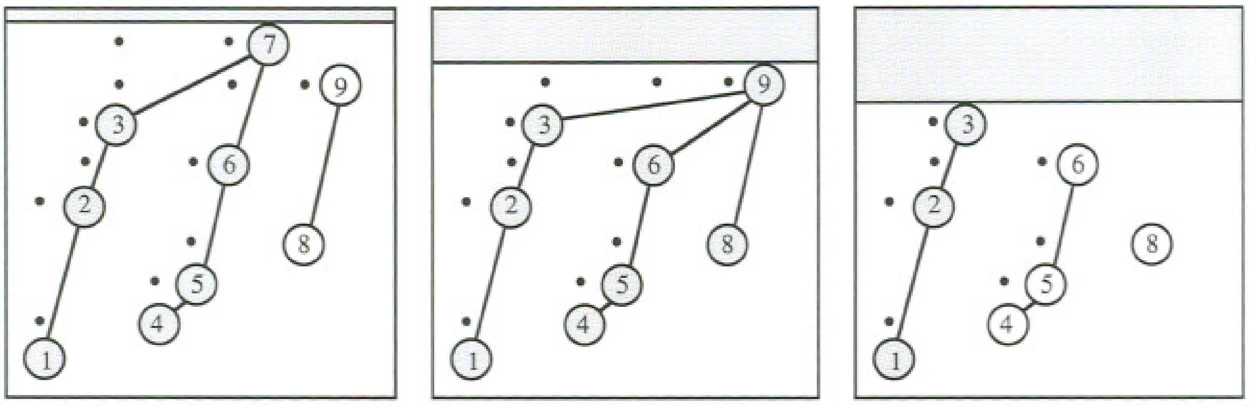

Fig. 5: The decreasing-sequence trees T[π_1], T[π_2] and T[π_3] of the permutation π=[7, 9, 3, 6, 2, 8, 5, 4, 1]. That is, the ds-trees T[7], T[9] and T[3].

Let T[π_i] be the *i*th ds-tree of a permutation π, $1 \leq i \leq n$. By definition, π_i is the root of the tree and π_j is a node iff $\pi_i > \pi_j$ and $i < j$. We can therefore compute the edge set of T[π_i] using the *D*-inversion matrix *D* of π as follows: Set <*D*[*k, i*], π_k> to be an edge of T[π_i] if $\pi_k < \pi_i$ and $\pi_k \in \{\pi_{i+1}, \pi_{i+2}, ..., \pi_n\}$.

Thus, the *D*-inversion matrix of a permutation π contains all the necessary information for constructing the ds-trees T[π_0], T[π_1] ..., T[π_n]. Sometimes, hereafter, we shall denote by T[π_0] the ds-tree T[π] rooted at $r = \pi_0$. We construct the ds-tree T[π_i] rooted at π_i by computing the parent function, $p(\pi_k)$, for each node π_k in [$\pi_{i+1}, \pi_{i+2}, ..., \pi_n$] such that $\pi_k < \pi_i$. Next, we list the parallel construction algorithm.

```
Algorithm ds-Trees (Decreasing-Subsequence-Trees):
Input  : A permutation $\pi$ on {1, 2, ..., n};
Output : The decreasing-subsequence trees T[$\pi_i$], $0 \leq i \leq n$;
begin
1. Compute the D-inversion matrix D of $\pi$;
2. Set $\pi_0$ to be the root of the tree T[$\pi_0$] and <$\pi_0$, $\pi_1$> to be an edge of T[$\pi_0$];
3. For every $\pi_k$ in {$\pi_1$, $\pi_2$, ..., $\pi_n$} do in parallel
       if D[k, 1] = $\infty$ then set $\pi_k$ to be a child of the root $\pi_0$; that is, $p(\pi_k) = \pi_0$
       else set $\pi_k$ to be a child of the node D[k, 1] $\neq \infty$; that is, $p(\pi_k)$ = D[k, 1];
4. For every i, $1 \leq i \leq n$, do in parallel
   4.1 Set $\pi_i$ to be the root of the tree T[$\pi_i$];
   4.2 For every $\pi_k$ in {$\pi_{i+1}$, $\pi_{i+2}$, ..., $\pi_n$} do in parallel
           if $\pi_k < \pi_i$ then set $\pi_k$ to be a child of the node D[k, i]; that is, $p(\pi_k)$ = D[k, i];
end;
```

In Fig. 6, we show the ds-trees T[π_0], T[π_1], ..., T[π_9] of the permutation π=[7, 9, 3, 6, 2, 8, 5, 4, 1], where $r = \pi_0$.

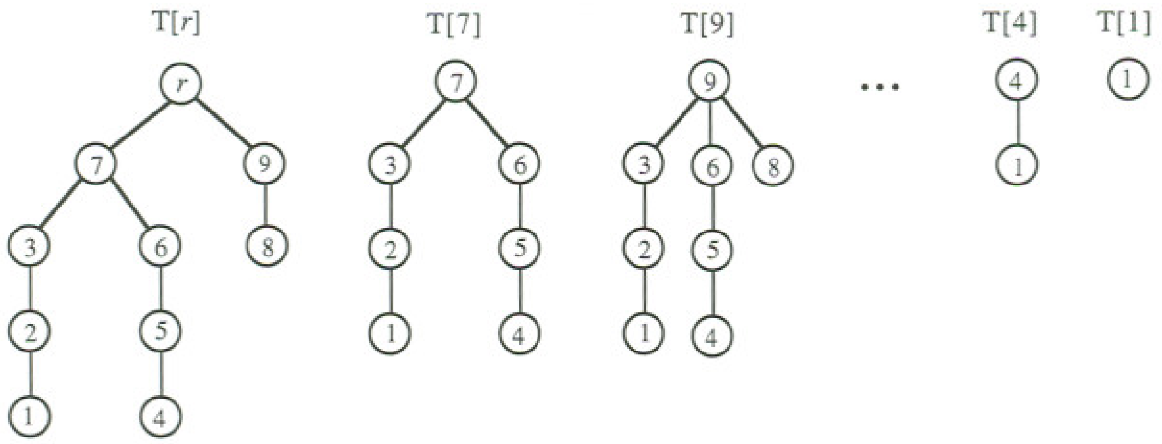

Fig. 6: The decreasing-subsequence trees $T[\pi_0]$, $T[\pi_1]$, ..., $T[\pi_9]$ of the permutation $\pi = [7, 9, 3, 6, 2, 8, 5, 4, 1]$. Here, $r = \pi_0$.

Remark 4.1 Let $T[\pi_i]$ be a ds-tree of a permutation π. By construction, the first decreasing subsequence of the sequence $[\pi_i, ..., \pi_n]$ forms a path from the root π_i to the leftmost leaf of the ds-tree $T[\pi_i]$, $0 \le i \le n$. Moreover, the elements of the first level of the ds-tree $T[\pi_i]$ form an increasing subsequence of the sequence $[\pi_{i+1}, ..., \pi_n]$, $0 \le i < n$.

4.2 The coloring-permutation tree $T^*[\pi]$

We are now in a position to developed a parallel algorithm for constructing a coloring-permutation tree $T^*[\pi]$. In particular, given a permutation $\pi = [\pi_1, ..., \pi_n]$, we formulate an algorithm that constructs a cp-tree $T^*[\pi]$ using the information contained in the ds-trees $T[\pi_0]$, $T[\pi_1]$, ..., $T[\pi_n]$.

Towards the construction of the cp-tree $T^*[\pi]$, we define two sets of nodes for each ds-tree $T[\pi_0]$, $T[\pi_1]$, ..., $T[\pi_n]$. For the ds-tree $T[\pi_i]$, $0 \le i \le n$, these two sets are the following:

link-nodes($T[\pi_i]$):	it contains all the nodes u of $T[\pi_i]$ having the following property: there exists a node v in $T[\pi_i]$ such that: (i) $level(v) = level(u)$, (ii) v lies on the left of u, and (iii) v is inverted by u; that is, $\pi_v^{-1} > \pi_u^{-1}$;
active-link-nodes($T[\pi_i]$):	it contains all the nodes u of *link-nodes*($T[\pi_i]$) having the property: there exists no node v in *link-nodes*($T[\pi_i]$) such that v inverts u.

In this paper, we assume that the above sets are ordered sets. That is, the elements in each set are arranged in the same order as they appear in π. The following algorithmic schemes describe the computations of the node sets *link-nodes*($T[\pi_i]$) and *active-link-nodes*($T[\pi_i]$), $0 \le i \le n$.

Computation of the vertex set link-nodes($T[\pi_i]$), $0 \le i \le n$: Let L_h be an array containing the nodes of the level h of $T[\pi_i]$ and let IL_h be an array containing the corresponding indices of the nodes of L_h in the permutation π. We assume that the nodes in L_h are arranged in the same order as they appear (from left to right) in the hth level.

The computation of the set *link-nodes*(T[π_i]) of the ds-tree T[π_i], $0 \le i \le n$, can be implemented as follows:

```
for every level $h$ of T[$\pi_i$], $1 \le i \le n$, do in parallel
1.  Let $L_h$ be the array containing the nodes of the tree at level $h$.
    Compute the array $IL_h$ having the property: the $k$th element of $IL_h$
    is the index of the $k$th element of $L_h$ in the permutation $\pi$;
2.  Compute the prefix maxima $p\text{-}max\text{-}IL_h$ of the array $IL_h$;
3.  For every node $u$ in $L_h$, do in parallel
        if $u$ is the $k$th element of $L_h$ then
            if $IL_h(k) \neq p\text{-}max\text{-}IL_h(k)$ then link-nodes(T[$\pi_i$]) $\leftarrow u$;
end;
```

We shall refer to the above algorithmic scheme as ***Scheme-A***. Let T[π_i] be a ds-tree and let *link-nodes*(L_h) be the set of the link nodes of the hth level, where $h \ge 1$. Fig. 7 shows the computation of the set *link-nodes*(L_h) using the algorithmic Scheme-A.

index	1	2	3	4	5	6	7	8	9
permutation	7	9	3	6	2	8	5	4	1

L_h	4	1	2	8	7	9
IL_h	8	9	5	6	1	2
$p\text{-}max\text{-}IL_h$	8	9	9	9	9	9

$\Rightarrow$ *link-nodes*(L_h) = [2, 8, 7, 9]

Fig. 7: The computation of the set *link-nodes*(L_h) of a tree T[π_i] with h-level nodes L_h = [4, 1, 2, 8, 7, 9].

Computation of the vertex set active-link-nodes(T[π_i]), $0 \le i \le n$: By definition, the vertex set *active-link-nodes*(T[π_i]) contains no two elements - say u and v - such that u inverts v (or v inverts u) in π. Thus, we simply have to remove all the elements v of the set *link-nodes*(T[π_i]) that are inverted by an element $u \in$ *link-nodes*(T[π_i]). This computation can be done using a similar technique as in the computation of the set *link-nodes*(T[π_i]). Specifically, it can be done using prefix maxima on the elements of the set *link-nodes*(T[π_i]); We shall refer to the algorithmic scheme that computes the active nodes of a ds-tree as ***Scheme-B***.

Having computed the sets *link-nodes*(T[π_i]) and *active-link-nodes*(T[π_i]) of the tree T[π_i], $1 \le i \le n$, let us now formulate a parallel algorithm for constructing the coloring-permutation tree T*[π_0]. The algorithm proceeds as follows:

```
Algorithm cp-Tree (Coloring-Permutation-Tree):
Input  : A permutation $\pi$ on $\{1, 2, ..., n\}$;
Output : The coloring-permutation tree $T^*[\pi_0]$;
begin
1. Construct the ds-trees $T[\pi_0]$, $T[\pi_1]$, ..., $T[\pi_n]$ and make all of them to be "active" trees;
2. For every active tree $T[\pi_i]$, $0 \le i \le n$, do in parallel
   2.1  Compute the level $level(v)$ of each node $v$ of $T[\pi_i]$;
   2.2  Compute the set $link\text{-}nodes(T[\pi_i])$ of the link nodes of $T[\pi_i]$;
   2.3  Compute the set $active\text{-}link\text{-}nodes(T[\pi_i])$ of the active link nodes of $T[\pi_i]$;
   2.4  If $active\text{-}link\text{-}nodes(T[\pi_i]) = \varnothing$, then make the tree $T[\pi_i]$ to be "non-active" tree;
3. Compute the set $link\text{-}nodes \leftarrow \cup_{0 \le i \le n}\, link\text{-}nodes(T[\pi_i])$;
4. For every node $u \notin link\text{-}nodes$ do in parallel
        Make the tree $T[u]$ to be "non-active" tree;
5. For every active tree $T[\pi_i]$, $0 \le i \le n$, do in parallel
   5.1  For every node $u \in active\text{-}link\text{-}nodes(T[\pi_i])$ do in parallel
            Copy the tree $T[u]$, and replace the subtree of $T[\pi_i]$ rooted at $u$ with
            the copy of $T[u]$;
   5.2  For every node $y$ of $T[\pi_i]$ such that $u$ inverts $y$ do in parallel
            Remove the subtree of $T[\pi_i]$ rooted at $y$ from the tree $T[\pi_i]$;
6. If $T[\pi_0]$ is an "active" tree, then execute step 2; otherwise return($T[\pi_0]$);
end;
```

Before proving the correctness of the algorithm cp-Tree, we should define the operations involved in step 5. Let $T[v_1]$ be a tree with n nodes $v_1, v_2, ..., v_n$ and let $preord(v_i)$ and $info(v_i)$ be the preorder number and the information of the node v_i, respectively. ***Copy*** the tree $T[v_1]$ is defined to be the operation that constructs a tree $T[u_1]$ with n nodes $u_1, u_2, ..., u_n$ such that (i) $preord(u_i) = preord(v_i)$, and (ii) $info(u_i) = info(v_i)$ for $i = 1, 2, ..., n$. Let $T[v]$ and $T[u]$ be two trees rooted at v and u, respectively, and let u be a node of $T[v]$. ***Replace*** the subtree T' rooted at u of the tree $T[v]$ with the tree $T[u]$ is defined to be the operation that makes the parent of the node u of $T[v]$ to point the root of the tree $T[u]$. ***Remove*** a subtree T' of a tree $T[v]$ is defined to de the operation that makes the subtree T' to be an empty tree.

5. Correctness

In this section we prove the correctness of the algorithm cp-Tree. We first establish the following characteristics used in the algorithm.

Lemma 5.1 Let $T[\pi_i]$ be the ith ds-tree of π, end let x be a link-node of $T[\pi_i]$ at level h. Let y be a node at level h-k such that x inverts y, $0 \le k \le h$-1. Then, there exist a node x' in the path from the root π_i to node x having the following properties:

(i) level(x') = level(y).

(ii) x' inverts y.

Proof. Let x, y be nodes of the tree $T[\pi_i]$ such that $level(x) = h$, $level(y) = h$-k and x inverts y. Let $P = [\pi_i, ..., x]$ be the path from the root π_i to node x. Obviously, there exists a node x' in P such that *level*(x') = h-k; see Fig. 8(a). Since x inverts y, it follows that y is not a node of P, $x > y$ and $\pi_x^{-1} < \pi_y^{-1}$. Moreover, $x' > x$ and $\pi_{x'}^{-1} < \pi_x^{-1}$. Hence, $x' > y$ and $\pi_{x'}^{-1} < \pi_y^{-1}$. Thus, x' inverts y. ☐

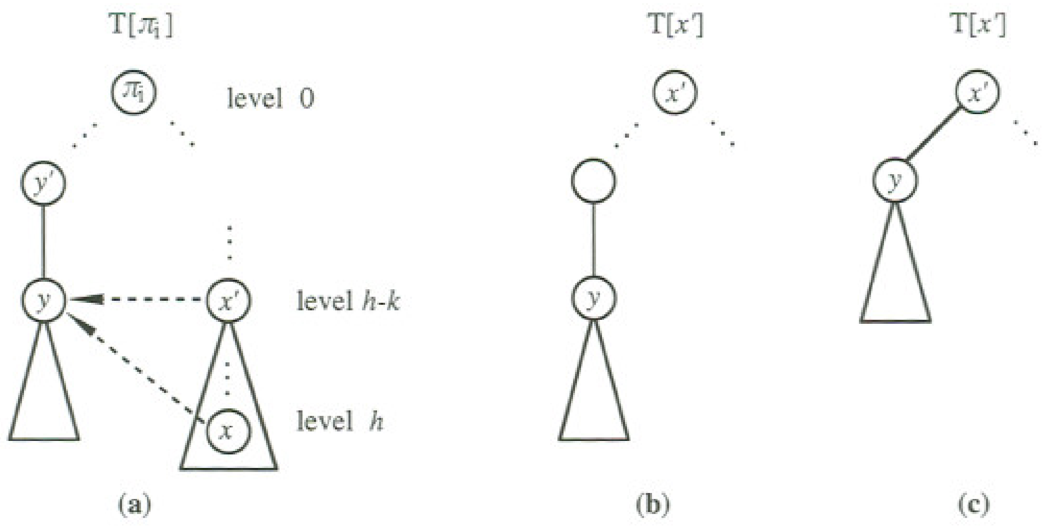

Fig. 8: (**a**) The tree T[π_i] and the inversion relationship of the nodes x, x' and y; (**b**) The tree T[x'] in the case where there exists a node x in *subtree*(π_i; x') such that x inverts y; (**c**) The tree T[x'] in the case where either there exists no node x in *subtree*(π_i; x') such that x inverts y or *subtree*(π_i; x') is empty.

Let T[π_i] be a ds-tree of a permutation π and let x be an internal node of T[π_i]. Hereafter, by *subtree*(π_i; x) we denote the subtree of T[π_i] rooted at x. Let *subtree*(π_i; x), *subtree*(π_j; x) be subtrees of the ds-trees T[π_i] and T[π_j], respectively. We say that these two subtrees are *equal*, denoted *subtree*(π_i; x) = *subtree*(π_j; x), if the one subtree is a copy of the other subtree (see Section 4 for the definition of the copy operation).

Lemma 5.2 Let x, y be nodes of a ds-tree T[π_i] such that x inverts y and let x' be a node in the path $P = [\pi_i, ..., x]$ such that level(x') = level(y) and x' inverts y. Let *subtree*(π_i; y) be a subtree of the ds-tree T[π_i] rooted at y. Then, *subtree*(π_i; y) = *subtree*(x'; y).

Proof. Since x' inverts y, it follows that $y < x'$ and $\pi_{x'}^{-1} < \pi_y^{-1}$. Moreover, there exists a node y' such that $y' > y$ and $\pi_{y'}^{-1} > \pi_{x'}^{-1}$, and y' is the parent of y'. It is easy to see that $y' < x'$; specifically, y' is the smallest element in the range $\pi_1^{-1}.. \pi_{x'}^{-1}$, which is larger than y. Let z be an arbitrary node of the *subtree*(y) of T[π_i]. Obviously, $z < y$ and $\pi_z^{-1} > \pi_y^{-1}$. We shall prove that z is also a node of the *subtree*(y) of T[x']. Suppose the contrary. Then, there exist a node y'' in the range $\pi_{x'}^{-1}.. \pi_y^{-1}$, such that $y'' < y$ and $y'' > z$. In this case, nodes y'' and y have parent the node y' in the ds-tree T[π_i]. Moreover, it is easy to see that y is a sibling of y''. Since $y'' > z$, it follows that z is a descendant of y''. Thus, z is not a node of the *subtree*(y) of T[π_i]; a contradiction. ☐

Lemma 5.3 Let x, y be nodes of a ds-tree T[π_i] such that x inverts y and let x' be a node in the path $P = [\pi_i, ..., x]$ such that level(x') = level(y) and x' inverts y. If a node w of *subtree*(π_i; y) is a link-node in T[π_i], then w is also a link-node in T[x'].

Proof. It follows immediately from the Lemma 5.2. ☐

Lemma 5.4 Let x, y be nodes of a ds-tree T[π_i] such that x inverts y and let x' be a node in the path $P = [\pi_i, ..., x]$ such that level(x') = level(y) and x' inverts y. Let *subtree*(π_i; y) and *subtree*(π_i; x') be two subtrees in T[π_i]. If node w is an active link-node in T[x'], then w is either a link-node of *subtree*(π_i; y) or a node of *subtree*(π_i; x') such that w inverts y.

Proof. By Lemma 5.2, the subtree *subtree*(π_i; y) of the tree T[π_i] is also a subtree *subtree*(x'; y) of the tree T[x']. Let level(y) and level(x) be the levels of the nodes y and x in the tree T[x'], respectively. We distinguish two cases. (I) level(y) > level(x). In this case w is obviously an active link-node in the ds-tree T[x']. (II) level(y) ≤ level(x). Then, by Lemma 5.1, there exist node w in the path $P = [x', ..., x]$ such that level(w) = level(y) and w inverts y. Since y inverts all the nodes in its subtree, it follows that w is a active link-node of the tree T[x']. □

Lemma 5.5 Let x_1, y_1, ..., z_1 be link-nodes of the tree T[π_i] such that $x_1 \in$ *active-link-nodes*(T[π_i]) and let w, ..., z_1 be link-nodes of the tree T[x_1] such that $w \in$ *active-link-nodes*(T[x_1]). After performing step 5 in the algorithm cp-Tree, $w \in$ *active-link-nodes*(T[π_i]).

Proof. By Lemma 5.4, w is either the node y_1 (see case I) or a node that inverts y_1, ..., z_1 (see case II). After performing step 5.1 in the algorithm cp-Tree, the tree T[x_1] is a subtree of the T[π_i]; that is, T[x_1] = *subtree*(π_i; x_1). After performing step 5.2, the subtree of T[π_i] rooted at y; that is, the *subtree*(π_i; y), is removed from the tree T[π_i] (note that x_1 inverts y). Thus, $w \in$ *active-link-nodes*(T[π_i]). □

Lemma 5.6 Let T[π_i] be a ds-tree of a permutation π and let w be a node such that $x \notin$ *link-nodes*(T[π_i]). After the kth iteration of step 5 of the algorithm cp-Tree if $x \in$ *link-nodes*(T[π_i]), then after the (k-1)th iteration of step 5 of the algorithm cp-Tree there exist tree T[π_j] such that $x \in$ *link-nodes*(T[π_j]) , where $j > i$.

Proof. It follows immediately from the Lemma 5.5. □

Let x, y, z be nodes of a ds-tree. Hereafter, x *inverts* y *inverts* z = (x *inverts* y) and (y *inverts* z). Fig. 9 shows a ds-tree, say T[π_i], in which the link nodes x_1, y_1, ..., z_1 have the property that x_1 *inverts* y_1 *inverts* ... *inverts* z_1. In this case we say that the tree T[π_i] has *single* link nodes. More precisely, we say that a ds-tree has *single* link nodes, if it has only one active link-node in each iteration of the step 5 of the algorithm cp-Tree.

Suppose that the ds-trees T[π_0], T[π_1], ..., T[π_n] of a permutation π on N_n have *single* link nodes. Based on the results of Lemmas 5.1 through 5.6, it is easy to see that there exist a sequence T_0 = [T[π_i], ..., T[π_j]] of length k having the following property:

$\pi_i \in$ *active-link-nodes*(T[π_0]),

and

$\pi_p \in$ *active-link-nodes*(T[π_{p-1}]), $p = i+1, ..., j$.

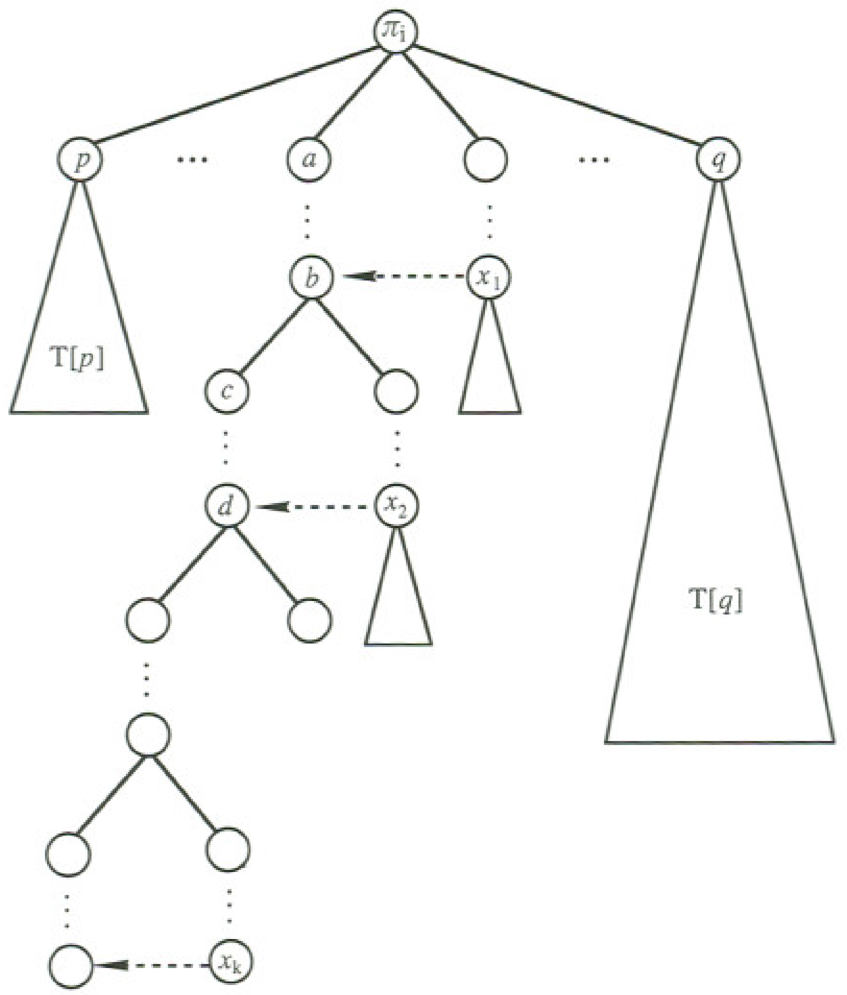

Fig. 9: The structure of the link-nodes of a ds-tree T[π_i] of a permutation π, having single link-nodes. Subtrees T[p] and T[q] have no link-nodes.

Consider now the following algorithmic scheme:

```
for $p = i, i+1, ..., j$ do
    Copy the tree T[$\pi_p$], and replace the subtree of T[$\pi_0$] rooted at $\pi_p$ with
    the copy of T[$\pi_p$];
    Remove the subtree subtree($\pi_0$; $y$), where $y$ is a node such that $\pi_p$ inverts $y$;
end;
```

The above algorithmic scheme constructs the tree T*[π_0]. We shall refer to this scheme as ***Scheme-C***; It is easy to see that the parallel implementation of Scheme-C corresponds to the step 5 of the algorithm cp-Tree.

Lemma 5.7 Let T[π_i] be a ds-tree of a permutation π and let x be its active link-node. Let y be the active link-node of the ds-tree T[x]. Let x be at level $h > 0$ in the tree T[π_i] and let y be at level $h > 0$ in the tree T[x]. Then, after the execution of the step 5 of the algorithm cp-Tree:

(i) node x is not an active link node of T[π_i].

(ii) the active link node of T[π_i] is the node y at level $2h$.

Thus, the algorithm cp-Tree produces the tree $T^*[\pi_0]$ after performing $O(\log k)$ times the steps 2 through 5, where k is the number of trees in the sequence T_0;

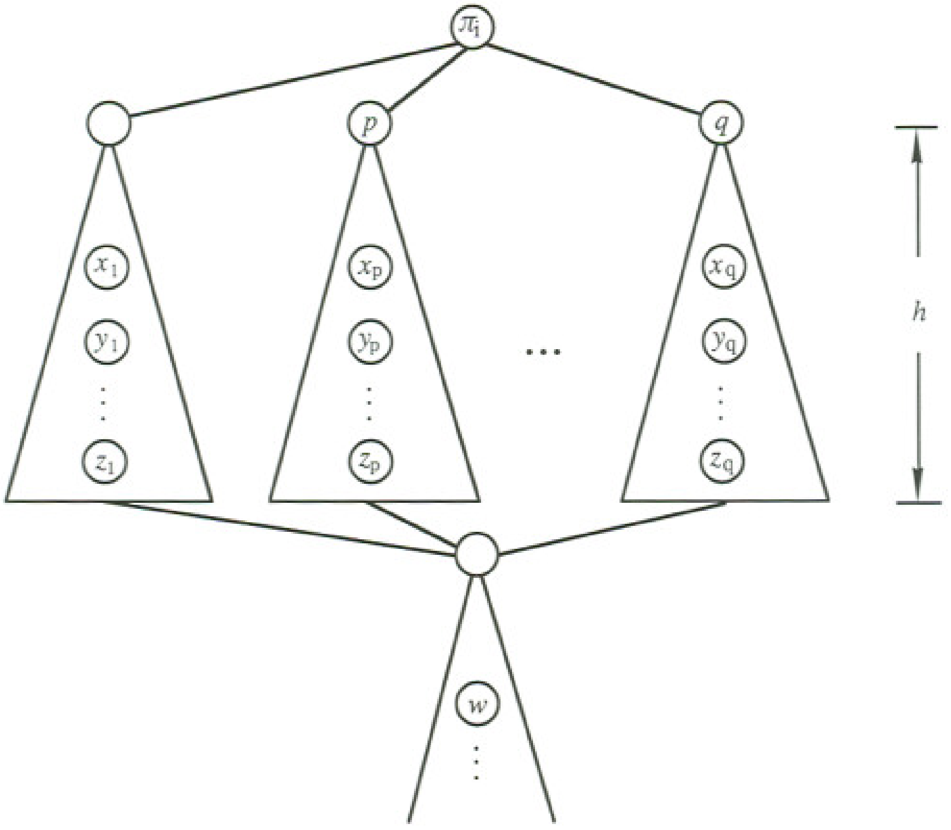

Fig. 10: The structure of the link-nodes of a ds-tree $T[\pi_i]$ of a permutation π having multiple link nodes.

Let $T[\pi_i]$ be a ds-tree of a permutation π. It is likely to have the case where there exist subtrees *subtree*$(\pi_i; p)$ and *subtree*$(\pi_i; q)$, where $i < p < q$, having the following property: there is no node in *subtree*$(\pi_i; q)$ which inverts a node in *subtree*$(\pi_i; p)$. Therefore, there are two active link nodes in the tree $T[\pi_i]$ - say x_p and x_q - since x_p (resp. x_q) is a link node in $T[\pi_i]$ and there is no node in $T[\pi_i]$ which inverts x_p (resp. x_q). In this case we say that the tree $T[\pi_i]$ has *multiple* link nodes. The structure of a ds-tree having multiple link nodes is shown in Fig. 10.

It is easy to see that any operation (see step 5 of the algorithm) on the nodes of *subtree*$(\pi_i; p)$ does not affect the link-node relationship of the nodes in *subtree*$(\pi_i; q)$. Thus, the results of this section can be summarized in the following Theorem.

Theorem 5.1 Given a permutation π on N_n, the algorithm **`cp-Tree`** constructs the coloring-permutation tree $T^*[\pi]$ after $O(\log n)$ iterations of the steps 2 through 5.

6. Resource Requirements

To establish the time and processor complexity of the algorithms we developed so far we shall use the well-known Concurrent-Read, Exclusive-Write PRAM model of parallel computation (CREW PRAM) [7, 12]. In this model, the operations of union ($\cup$), intersection ($\cap$) and subtraction (-) on n

elements are executed in $O(\log n)$ time with $O(n / \log n)$ processors. The prefix sums of n elements can also be computed within the same time and processor complexities. Moreover, in this model, the computation of the postorder, preorder and inorder numbers of each node of a tree, as well as the level of each node of a tree can be done in $O(\log n)$ time with $O(n / \log n)$ processors using the well-known Euler-tour technique (actually, all the above operations are computed in the EREW PRAM model within the same time-processor complexity); see [7].

6.1 The D-inversion-matrix algorithm

We first compute the time and processor complexity of the algorithm for the computation of the D-inversion matrix of a permutation π on N_n. Obviously, substeps 1.1 through 1.3 are executed in $O(1)$ sequential time. Substep 1.4 computes the suffix minima of an array of length i, where $1 \leq i \leq n$. It is well-known that the suffix minima of n elements can be computed in $O(\log n)$ time using $n / \log n$ processors on the EREW PRAM model (Note that the suffix minima problem is based on the computation of the prefix-sums of n elements.) [7]. The four substeps are executed for every i, $1 \leq i \leq n$, in parallel. Thus, the following theorem holds.

Theorem 6.1 Given a permutation π on N_n, the algorithm `D-inversion-matrix` computes the D-inversion matrix of a permutation π in $O(\log n)$ time using $n^2 / \log n$ processors on the CREW PRAM model.

6.2 The ds-Trees algorithm

Let us now compute the overall complexity of the algorithm `ds-Trees` which constructs the $n+1$ decreasing-subsequence trees $T[\pi_0]$, $T[\pi_1]$, ..., $T[\pi_n]$ of a permutation π.

The algorithm consists of 4 steps: *Step 1*. By Theorem 6.1, the computation of the D-inversion matrix of a permutation π on n elements can be done in $O(\log n)$ time using $n^2 / \log n$ processors. *Step 2*. Obviously, it takes $O(1)$ sequential time. *Step 3*. Having computed the D-inversion matrix, this step requires $O(1)$ time and n processors or $O(\log n)$ time and $n / \log n$ processors. *Step 4*. The time and processor requirement of the substeps 4.1 and 4.2 are the same as that required for steps 2 and 3, respectively. Both substeps are executed for every i, $1 \leq i \leq n$, in parallel. Therefore, step 4 requires $O(1)$ time and n^2 processors or $O(\log n)$ time and $n^2 / \log n$ processors.

Take into consideration the time and processor complexity of each step of the algorithm, we can present the following result.

Theorem 6.2 Given a permutation π on N_n, the algorithm `ds-Trees` constructs the decreasing-subsequence tree of the permutation π in $O(\log n)$ time using $n^2 / \log n$ processors on the CREW PRAM model.

6.3 The cp-Tree algorithm

Next, we compute the time and processor complexity of the algorithm `cp-Tree`. We shall obtain the overall complexity by computing the complexity of each step separately. Let us first compute the time and processor complexity of some operations used in the algorithm.

We have mentioned that the level of each node of a tree can be computed in $O(\log n)$ time using $O(n / \log n)$ processors on the EREW PRAM model [7]. Let us now compute the complexity of the algorithmic scheme which computes the link nodes of a ds-tree (see Scheme-A). In this scheme, the array L_h stores the nodes of the level h of a ds-tree as they appear in the tree from left to right. It is easy to see that the array IL_h which contains the indices of the nodes of L_h in the permutation π can be computed in $O(\log n)$ time using $n_h / \log n$ processors on the EREW PRAM model, where n_h is the number of nodes of level h. The prefix minima of n elements can be computed in $O(\log n)$ time using $n_h / \log n$ processors on the EREW PRAM model [7]. Therefore, the execution of Scheme-A requires $O(\log n)$ time and $n / \log n$ processors on the EREW PRAM model. In a similar way we show that the Scheme-B, which computes the active link nodes of a ds-tree, is executed in $O(\log n)$ time using $n / \log n$ processors on the EREW PRAM model. Thus, the following lemma holds.

Lemma 6.1 The sets of link and active nodes and the level of each node of a ds-tree with n nodes can be computed in $O(\log n)$ time using $O(n / \log n)$ processors on the EREW PRAM model.

It is easy to see that the Lemma 6.1 gives us the complexity of the step 2 of the algorithm cp-Tree in the case where it is executed for a ds-tree $T[\pi_i]$, $0 \le i \le n$.

We focus now on step 5, which is the crucial step for the processor complexity of the algorithm cp-Tree. Obviously, a ds-tree $T[u]$ with n nodes can be copied in $O(\log n)$ time using $O(n / \log n)$ processors on the EREW PRAM model. The tree $T[u]$ is moved in a ds-tree $T[\pi_i]$ by simply make the parent of u in $T[\pi_i]$ to point in $T[u]$. This operation takes $O(1)$ sequential time. If h is the level of the node u in $T[\pi_i]$, then all the nodes at level h that are inverted by node u can be found in $O(\log n)$ time using $n_h / \log n$ processors on the EREW PRAM model, where n_h is the number of nodes of level h.

To compute the processor complexity of the algorithm cp-Tree, we first compute the processor complexity of step 5. We prove the following lemma.

Lemma 6.2 Each iteration of step 5 of the algorithm cp-Tree requires $O(n^2 / \log n)$ processors on the EREW PRAM model.

Proof. Let $T[\pi_0]$, $T[\pi_1]$, ..., $T[\pi_n]$ be the ds-trees of a of a permutation π. If all these trees have single link nodes, then we are done since for every tree $T[\pi_i]$ at most one tree $T[\pi_j]$ is copied and moved in $T[\pi_i]$, where $i < j$ and $0 \le i \le n$-1. We now consider the case where the ds-trees have multiple link nodes. Let $T[\pi_i]$ be such a tree having k active link nodes during an iteration of step 5 and let $T[\pi_i]$ be an active ds-tree. It follows that k trees are copied and moved in $T[\pi_i]$. Therefore, step 5 requires $kn / \log n$ processors for a ds-tree. Let $T[x_1]$, $T[x_2]$, ..., $T[x_p]$ be the active ds-trees during an iteration of step 5 and let $k_1, k_2, ..., k_p$ be the numbers of active link nodes in these ds-trees. Then, $(k_1 + k_2 + ... + k_p)n / \log n$ processors must be available for the execution of step 5. Obviously, if x, y are two active link nodes of an active ds-tree, then $x \ne y$. Let x be an active link node of an active ds-trees $T[x_i]$, $1 \le i \le p$-1. Assume that x is also an active link node of another active ds-tree $T[x_j]$, where $i < j$ and $2 \le i \le p$. Then, x_j is a node of the ds-tree $T[x_i]$. Since x is an active link node in $T[x_i]$, x_j is an ancestor of the node x of $T[x_i]$. Thus, $x_j \notin$ *link-nodes* (see step 4). It follows that the ds-tree $T[x_j]$ is a non-active tree, a contradiction. Then, we conclude that if x, y are two active link nodes in the forest of the active ds-trees, then $x \ne y$. Thus, $(k_1 + k_2 + ... + k_p) \le n$ and the lemma holds. □

We are now ready to compute the time and processor complexity of the algorithm cp-Tree. The algorithm consists of six steps. *Step 1*. By Theorem 6.2, the construction of the ds-trees of a permutation π on N_n can be done in $O(\log n)$ time using $n^2 / \log n$ processors on the CREW PRAM model. *Step 2*. This step incorporate operations whose time and processor complexity are given by Lemma 6.1. Thus, the step is executed in $O(\log n)$ time with $n^2 / \log n$ processors on the EREW PRAM model. *Step 3*. The union of $n+1$ sets, each of length $O(n)$, is performed in $O(\log n)$ time with $n^2 / \log n$ processors on the EREW PRAM model. *Step 4*. Obviously, this step is executed in $O(1)$ with n processors on the EREW PRAM model. *Step 5*. It consists of substeps 5.1 and 5.2, which are executed for every active tree $T[\pi_i]$, $0 \leq i \leq n$. *Substep 5.1*. Lemma 6.2 coupled with the discussion concerning the time complexity of the operations of copy and move a ds-tree implies that substep 5.1 is executed in $O(\log n)$ time using $n^2 / \log n$ processors on the CREW PRAM model. *Substep 5.2*. The time and processor requirement of this substep is the same as that required for substep 2.2. *Step 6*. Obviously, this step is executed in $O(1)$ sequential time.

By Theorem 5.1, the algorithm cp-Tree performs $O(\log n)$ iteration. Thus, take into consideration the time and processor of each step of the algorithm, we can present the following result.

Theorem 6.3 Given a permutation π on N_n, the coloring-permutation tree $T^*[\pi]$ can be constructed in $O(\log^2 n)$ time using $O(n^2 / \log n)$ processors on the CREW PRAM model.

6.4 The Coloring algorithm

The algorithm `Coloring` incorporate all the operation described in the previous algorithms. More precisely, by Theorem 6.3 the step 1 takes $O(\log^2 n)$ time and $O(n^2 / \log n)$ processors on the CREW PRAM model. The level of each node of tree with n nodes is computed in $O(\log n)$ time with $n / \log n$ processors on the EREW PRAM model. Finally, the step 3 of the algorithm requires $O(1)$ time and $O(n)$ processors. Thus, we obtain the following theorem.

Theorem 6.4 The problem of coloring permutation graphs can be solved in $O(\log^2 n)$ time using $O(n^2 / \log n)$ processors on the CREW PRAM model.

7. Conclusions

In this paper we studied the problem of coloring permutations graphs using the Lattice representation of a permutation [13] and the relationship between permutations and binary search trees. We proposed an efficient parallel algorithm which colors a permutation graph in $O(\log^2 n)$ time using $O(n^2 / \log n)$ processors on the CREW PRAM model, where n is the number of vertices in the permutation graph.

The idea of our algorithm is motivated by the work performed by C-W Yu and G-H Chen [19]. They presented an algorithm which takes as input a permutation graph and transforms it into a set of planar points, constructs an acyclic directed graph, and finally solves the largest-weight path problem on this acyclic digraph. Using this strategy, they solve the coloring problem on permutation graph in $O(\log^2 n)$ time using $O(n^3 / \log n)$ processors on a CREW PRAM model of computation or in $O(\log n)$ time using $O(n^3)$ processors on a CRCW PRAM [2, 7, 12]. The approach used in this paper is different from the previous algorithm. We presented an algorithm which takes as input a permutation π, and constructs the color tree $T^*[\pi]$ using combinatorial properties on π.

Then, it solves the coloring problem for the permutation graph $G[\pi]$ by computing the level of each node of the tree $T^*[\pi]$. Our parallel algorithm improves in performance upon the best-known parallel algorithms for the same problem.

In closing, we should point out that with slight modifications our coloring algorithm can also solve the weighted clique problem, the weighted independent set problem, the clique cover problem, and the maximal layers problem within the same complexity bounds [19].

References

[1] M.J. Atallah, G.K. Manacher and J. Urrutia, Finding a minimum independent dominating set in a permutation graph, *Discrete Applied Mathematics*, vol. 21, pp. 177-183, 1988.

[2] P. Beame and J. Hastad, Optimal bounds for decision problems on the CRCW PRAM, *J. Assoc. Comput. Mach.*, vol. 36. pp. 643-670, 1989.

[3] A. Brandstadt and D. Kratsch, On domination problems for permutation and other graphs, *Theoretical Computer Science*, vol. 54, pp. 181-198, 1987.

[4] M. Farber and J.M. Keil, Domination in permutation graphs, *Journal of Algorithms*, vol. 6, pp. 309-321, 1985.

[5] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, Inc., 1980.

[6] D. Helmbold and E.W. Mayr, Applications of parallel algorithms to families of perfect graphs, *Computing*, vol. 7, pp. 93-107, 1990.

[7] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Inc., 1992.

[8] T.R. Jensen and B. Toft, *Graph Coloring Problems*, John Wiley & Sons, Inc., 1995.

[9] D. Kozen, U.V. Vazirani and V.V. Vazirani, NC algorithms for comparability graphs, interval graphs, and testing for unique perfect matching, *Lecture Notes in Computer Science*, vol. 206, pp. 498-503, 1985.

[10] J.Y.-T. Leung, Fast algorithms for generating all maximal independent sets of interval, circular-arc and chordal graphs, *Journal of Algorithms*, vol. 5, pp. 22-35, 1984.

[11] A. Pnueli, A. Lempel and S. Even, Transitive orientation of graphs and identification of permutation graphs, *Canadian J. Math.*, vol. 23, pp. 160-175, 1971.

[12] J. Reif (editor), *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1993.

[13] R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Inc., 1996.

[14] J. Spinrad, On comparability and permutation graphs, *SIAM Journal on Computing*, vol. 14, pp. 658-670, 1985.

[15] J. Spinrad, A. Brandstadt and L. Stewart, Bipartite permutation graphs, *Discrete Applied Mathematics*, vol. 18, pp. 279-292, 1987.

[16] K.J. Supowit, Decomposing a set of points into chains, with applications to permutation and circle graphs, *Inform. Process. Lett.*, vol. 21, pp. 249-252, 1985.

[17] K.H. Tsai and W.L. Hsu, Fast algorithms for the dominating set problem on permutation graphs, *Lecture Notes in Computer Science: Algorithms*, vol. 450, pp. 109-117, 1990.

[18] S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirakawa, A new algorithm for generating all the maximal independent sets, *SIAM Journal on Computing*, vol. 6, pp. 505-517, 1977.

[19] C-W. Yu and G-H. Chen, Parallel algorithms for permutation graphs, *BIT*, vol. 33, pp. 413-419, 1993.

[20] C-W. Yu and G-H. Chen, Generate all maximal independent sets in permutation graphs, *Intern. J. Computer Math.*, vol. 47, pp. 1-8, 1993.