ELSEVIER

Discrete Applied Mathematics 120 (2002) 165–195

DISCRETE APPLIED MATHEMATICS

Coloring permutation graphs in parallel

Stavros D. Nikolopoulos

Department of Computer Science, University of Ioannina, P.O. Box 1186, GR-45110 Ioannina, Greece

Received 1 May 2000; received in revised form 8 April 2001; accepted 3 June 2001

Abstract

A *coloring* of a graph G is an assignment of colors to its vertices so that no two adjacent vertices have the same color. We study the problem of coloring permutation graphs using certain properties of the lattice representation of a permutation and relationships between permutations, directed acyclic graphs and rooted trees having specific key properties. We propose an efficient parallel algorithm which colors an n-node permutation graph in $O(\log^2 n)$ time using $O(n^2/\log n)$ processors on the CREW PRAM model. 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 coloring-permutation tree. © 2002 Elsevier Science B.V. All rights reserved.

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

1. Introduction

Let $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ be a permutation over the set $N_n = \{1, 2, \ldots, n\}$. The n-node graph $G[\pi] = (V, E)$ is defined as follows: $V = N_n$ and an edge $(i, j) \in E$ if and only if $(i - j)(\pi_i^{-1} - \pi_j^{-1}) < 0$, for all $i, j \in V$ where π_i^{-1} is the index of the element i in π. A graph G is a *permutation graph* if there exists a permutation π on N_n such that G is isomorphism to $G[\pi]$. The graph $G[\pi]$ is also known as the *inversion graph* of π. We, therefore, assume in this paper that a permutation graph $G[\pi]$ is represented by the corresponding permutation π; see [6].

Many researchers have devoted their work to the study of permutation graphs. They have proposed sequential and/or parallel algorithms for recognizing permutation graphs and solving combinatorial and optimization problems on them. For a sequential environment, Pnueli et al. [14] gave an $O(n^3)$ time algorithm for recognizing permutation graphs using the transitive orientable graph test. Later, Spinrad [17] improved their

E-mail address: stavros@cs.uoi.gr (S.D. Nikolopoulos).

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 [18], 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 [19] solved the coloring problem, the maximum clique problem, the clique cover problem and the maximum independent set problem, all in $O(n \log n)$ sequential time. Moreover, Farber and Keil [5] solved the weighted domination problem and the weighted independent domination problem in $O(n^3)$ time, using dynamic programming techniques. Later, Brandstadt and Kratsch [4] published an $O(n^2)$ time algorithm for the weighted independent domination problem. Atallah et al. [2] solved the independent domination set problem in $O(n \log^2 n)$ time, while Tsai and Hsu [20] 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. [21] 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 [11] 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 [23], Yu and Chen showed that all the maximal independent sets can be obtained in $O(n \log n + k)$ time using $O(n \log n)$ space. Recently, Nikolopoulos et al. [13] studied the behaviour of the on-line coloring algorithm *First-Fit* (*FF*) on the class of permutation graphs and proved that this class of graphs is not *FF* χ-bounded.

Although many sequential algorithms have been proposed for permutation graphs, few parallel algorithms have appeared in the literature. Due to the work of Helmbold and Mayr [7] and Kozen et al. [10], 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 the 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 the same model of computation. Moreover, given a permutation graph, their algorithm can construct the permutation that represents the permutation graph. For descriptions of various PRAM models of computation, see [3,8,15].

Our objective is to study the coloring problem on permutation graphs. Yu and Chen [22] proposed a technique that transfers the coloring problem into the largest-weight path problem. Their algorithm solves the coloring problem in $O(\log^2 n)$ time with $O(n^3/\log n)$ processors on the CREW PRAM, or in $O(\log n)$ time with $O(n^3)$ processors on the CRCW PRAM model. They also proposed parallel algorithms that solve the weighted clique problem, the weighted independent set problem, the clique cover problem, and the maximal layers problem with the same time and processor complexity. Recently, using a similar transformation technique Andreou and Nikolopoulos [1] designed a parallel algorithm which solves the problem of coloring a permutation graph of size n in $O(\log^2 n)$ time using $O(n^3/\log^3 n)$ processors on the CREW PRAM model of computation. Moreover, they showed that the coloring problem on permutation graphs can be solved in $O(\log n \log\log n)$ time in the average-case with $O(n^2)$ processors.

In this paper, we present a parallel algorithm for the problem of coloring a permutation graph, which runs in $O(\log^2 n)$ time with $O(n^2/\log n)$ processors on the CREW PRAM model. Our algorithm uses a strategy to transform a permutation graph G into a rooted tree T and solves the coloring problem on G by computing the vertex-level function on T. We design our algorithm using certain combinatorial properties of the lattice representation of a permutation [16] and relationships between permutations, directed acyclic graphs and rooted trees having specific key properties. Specifically, given a permutation π (or its corresponding graph $G[\pi]$), we first construct a rooted tree $T[\pi]$ by exploiting the inversion and d-inversion relations of the element of π and, then, from the tree $T[\pi]$ we construct a rooted tree $T^*[\pi]$ which we call *coloring-permutation tree* or *cp-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 cp-tree $T^*[\pi]$. We show that the cp-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 can be computed in $O(\log n)$ time with $O(n/\log n)$ processors on the EREW PRAM model using the well-known Euler-tour technique [8], 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 organized 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 cp-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 cp-tree. In Section 4, we present a parallel algorithm for the construction of the cp-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 7 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 colour. 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 the *chromatic number* of G are denoted by $\chi(G)$ [7,9].

Let π be a permutation over the set 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 $\pi_i > \pi_j$.

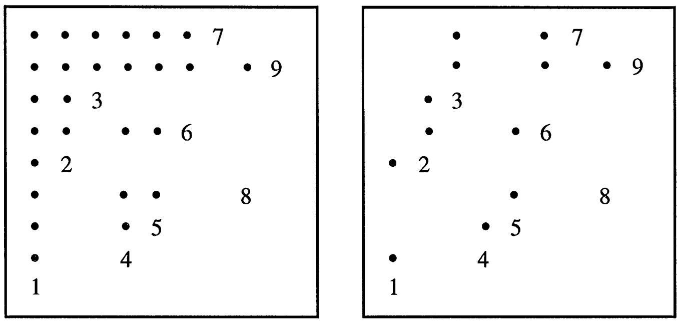


Fig. 1. Lattice representation of the permutation $\pi=(7,9,3,6,2,8,5,4,1)$ and its d-inversion relation of each pair of elements of π.

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. For example, in the permutation $\pi=(7,9,3,6,2,8,5,4,1)$, 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; see Chapter 6 in [16]; also in [2,4,5].

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, $1 \leqslant i \leqslant n$. 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 $(\pi_1,\pi_2,\dots,\pi_n)$ 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 elements of the permutation π. If π_i is below and π_j to the right, then π_j inverts π_i or, equivalently, π_i is inverted by π_j. In Fig. 1 (left 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 right lattice of the same figure.

We 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 node 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 node u. 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 few 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 relations 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 $i \geqslant 1$. That is, C_i contains all the nodes u of $T^*[\pi]$ such that *level*$(u)=i$, $1 \leqslant i \leqslant k$, where $k=\chi(G[\pi])$.

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

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

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

(iii) Let $T_1,T_2,\ldots,T_m$ $(m \geqslant 1)$ be the trees in $\mathscr{F}$, and let $\pi_{p(1)},\pi_{p(2)},\ldots,\pi_{p(m)}$ be the roots of those trees, respectively. Let r be a new node such that $r=n+1$. Then, construct a rooted tree $T[\pi]$ consisting of the nodes and edges of $T_1,T_2,\ldots,T_m$, the new node r, and the new edges $\langle r,\pi_{p(1)}\rangle,\langle r,\pi_{p(2)}\rangle,\ldots,\langle r,\pi_{p(m)}\rangle$. The root of $T[\pi]$ is r, and $T_1,T_2,\ldots,T_m$ 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 $P=(r,\pi_p,\ldots,\pi_q)$.

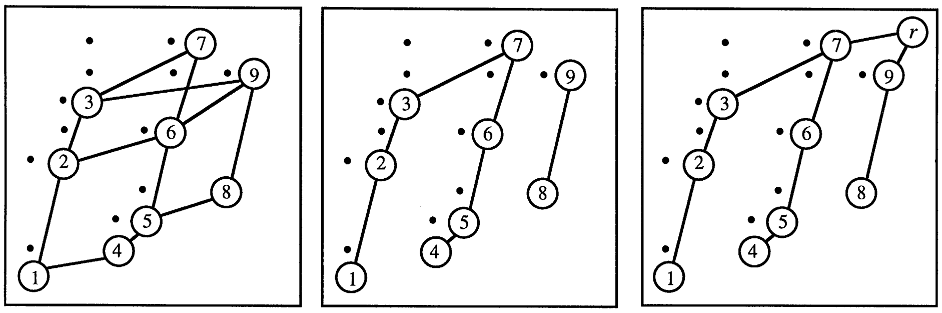


Fig. 2. The construction of the tree $T[\pi]$ of the permutation $\pi=(7,9,3,6,2,8,5,4,1)$.

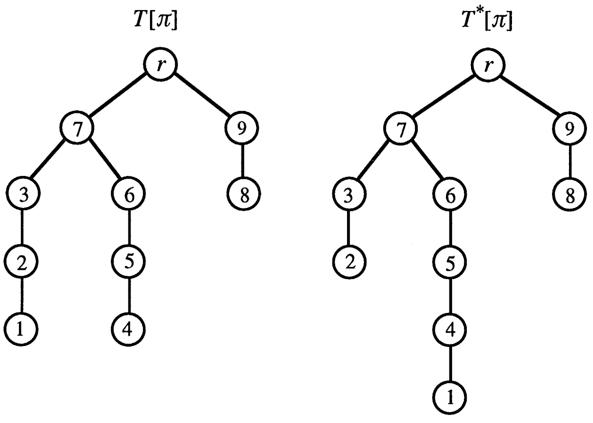


Fig. 3. The ds-tree $T[\pi]$ and a cp-tree $T^*[\pi]$ of the permutation $\pi=(7,9,3,6,2,8,5,4,1)$.

Moreover, if π_i and π_j are two elements of P such that $\pi_i > \pi_j$ and $p \leqslant i,j \leqslant q$, then $\pi_i^{-1} < \pi_j^{-1}$ in π. Based on this property we shall refer, hereafter, to the tree $T[\pi]$ as the *decreasing-subsequence tree*, or the *ds-tree* for short.

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

(i) $V^*=\{r,\pi_1,\ldots,\pi_n\}$, where r is the root of the tree; $r=n+1$;
(ii) if $\langle\pi_i,\pi_j\rangle \in E^*$, then π_i inverts π_j in π;
(iii) there is no pair of nodes π_i,π_j such that $level(\pi_i) \geqslant level(\pi_j)$ and π_i inverts π_j in π.

It is easy to see that the ds-tree $T[\pi]$ of a permutation π is a coloring-permutation tree $T^*[\pi]$, or *cp-tree* for short, if there are no nodes π_i,π_j in $T[\pi]$ such that $level(\pi_i) \geqslant level(\pi_j)$ and π_i inverts π_j in π. Fig. 3 shows the ds-tree $T[\pi]$ and a cp-tree $T^*[\pi]$ of the sample permutation π. In the same figure, $T[\pi]$ is not a cp-tree because there exists a pair of nodes $(4,1)$ such that $level(4) \geqslant level(1)$ and 4 inverts 1 in π.

Having constructed the ds-tree $T[\pi]$ of a permutation π, let us now show the way we can construct the cp-tree $T^*[\pi]$.

We first define a tree operation which is useful for showing the construction of a cp-tree $T^*[\pi]$ form the ds-tree $T[\pi]$. Let π_i, π_j be two nodes of $T[\pi]$ such that $level(\pi_i) \geqslant level(\pi_j)$ and π_i inverts π_j in π, and let $p(\pi_j)$ be the parent of the node π_j. Then, we define an operation which makes the subtree of node $p(\pi_j)$ rooted at π_j to be a subtree of node π_i. We call this operation *inversion-move operation* and we formally define it as follows:

Inversion-move operation: Let π_i, π_j be two nodes of the tree $T[\pi]$. Delete the edge $\langle p(\pi_j), \pi_j \rangle$ from and add a new edge $\langle \pi_i, \pi_j \rangle$ to $T[\pi]$ if $level(\pi_i) \geqslant level(\pi_j)$ in $T[\pi]$ and π_i inverts π_j in π; this operation is denoted by i-move(π_j, π_i).

Let T' be the resulting tree after applying the operation i-move(π_j, π_i) on $T[\pi]$. In T' the nodes π_i and π_j have the following properties: (i) $\langle \pi_i, \pi_j \rangle$ is an edge, (ii) π_i inverts π_j, and (iii) $level(\pi_i) < level(\pi_j)$ in T'. Thus, we can construct a cp-tree $T^*[\pi]$ by applying inversion-move operations on the tree $T[\pi]$ until no pair of nodes (π_i, π_j) remains in $T[\pi]$ such that $level(\pi_i) \geqslant level(\pi_j)$ and π_i inverts π_j in π.

For example, let $T[\pi]$ be the ds-tree of the sample permutation π used throughout the paper (see Fig. 3; leftmost tree). In this tree, 4 inverts 1 and $level(4) = level(1)$ in $T[\pi]$. It is easy to see that we can construct a cp-tree $T^*[\pi]$ by applying the inversion-move operation i-move$(1, 4)$ on the ds-tree $T[\pi]$ (see Fig. 3; rightmost tree).

By definition, the ds-tree $T[\pi]$ has the following structural property: the parent of a node π_p in $T[\pi]$ is the minimum π_q such that (a) π_q is inverted by r, and (b) π_q d-inverts π_p. We note that after applying an inversion-move operation on the ds-tree $T[\pi]$, this property no longer holds.

Remark 3.1. Let $G[\pi]$ be a permutation graph and let $T^*[\pi]$ be a cp-tree of $G[\pi]$ 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 $T[\pi]$, we conclude that if $P = (\pi_i, \ldots, \pi_j)$ is a path in $T^*[\pi]$, then the subgraph of $G[\pi]$ induced by the set $\{\pi_i, \ldots, \pi_j\}$ is an m-node complete graph, where m is the number of elements in P.

Remark 3.2. Let $T^*[\pi]$ be a cp-tree of $G[\pi]$ rooted at r and let $P = (r, \pi_i, \ldots, \pi_j)$ be a path from r to a node π_j. It is clear that the inversion-move operation has respect for the properties of the ds-tree $T[\pi]$. Thus, by construction, P forms a decreasing sequence $(r, \pi_i, \ldots, \pi_j)$.

We now show that there is a 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 $1 \leqslant i \leqslant k$. Recall that $k = \chi(G[\pi])$.

Lemma 3.1. *Let π be a permutation over the set 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. Corollary 7.4 in [6]. □

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. By definition, if $\langle \pi_i, \pi_j \rangle \in E^*$, then π_i inverts π_j in π. It follows that $i > j$ and $\pi_i > \pi_j$ in π. (see also Remark 3.2). □

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 $k \leqslant \chi(G[\pi])$. (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 $S = (\pi_p, \ldots, \pi_q)$ 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 $level(\pi_i) \geqslant level(\pi_j)$ 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 two nodes of the tree $T^*[\pi]$. If (π_i, π_j) is an edge in $G[\pi]$, then $level(\pi_i) \neq level(\pi_j)$ in $T^*[\pi]$.*

Proof. Suppose that $level(\pi_i) = level(\pi_j)$. Since π_i and π_j are adjacent in $G[\pi]$, it follows that π_i inverts π_j in π. Thus, there is a pair of nodes (π_i, π_j) in $T^*[\pi]$ such that $level(\pi_i) \geqslant level(\pi_j)$ 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 π and its corresponding graph $G[\pi] = (V, E)$;
Output: The color of each vertex $\pi_i \in V$, $i = 1, 2, \ldots, n$;

```
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 $T^*[\pi]$, $1 \leqslant i \leqslant n$;
3. Set $color(\pi_i) \leftarrow level(\pi_i)$, $i = 1, 2, \ldots, n$;
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 $T^*[\pi]$, $k \leqslant n$. 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 Theorem 3.1. Its proof relies on the results of Lemmas 3.3 and 3.4. Hence, we obtain the following result.

Theorem 3.1. *Let π be a permutation over the set N_n. 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 shown 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 $T^*[\pi]$.

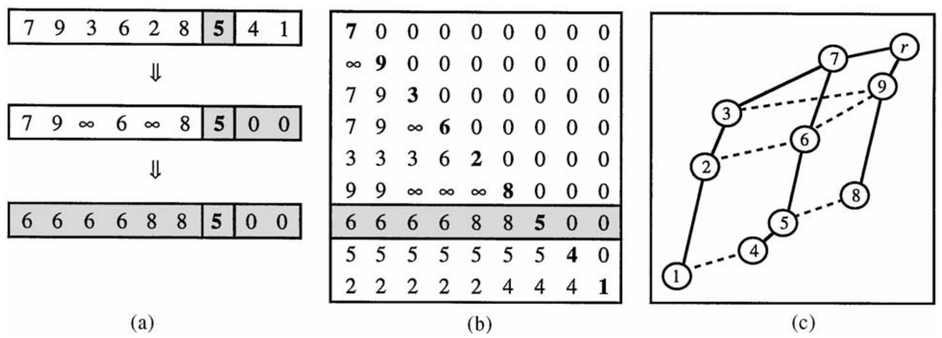
4.1. The decreasing-subsequence trees

As stated previously, the ds-tree $T[\pi]$ is constructed by exploiting the *d-inversion* relation on the permutation π. Obviously, the d-inversion set of an element is a subset of its inversion set. We can easily see that the inversion set *inversion-set*(π_i) 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; see also [2,4,5,16]. Thus, we can compute the *d-inversion-set* of an element π_i by computing the suffix minima $(\pi_{i1}, \pi_{i2}, \ldots, \pi_{i(i-1)})$ of the sequence $(\pi_1, \pi_2, \ldots, \pi_{i-1})$, where $\pi_k = \infty$ if $\pi_k \notin$ *inversion-set*(π_i), $1 \leqslant k \leqslant i-1$; that is, the element π_{ij} is the minimum element among $\{\pi_j, \pi_{j+1}, \ldots, \pi_{i-1}\}$, $1 \leqslant j \leqslant i-1$; see Fig. 4(a).

Having computed the *d-inversion-set* of each element π_i of a permutation π ($1 \leqslant i \leqslant n$), we can easily compute the *d-inversion matrix D* of π; D is an $n \times n$ matrix containing all the necessary information for the d-inversion relation of the permutation π.

Let $(\pi_{i1}, \ldots, \pi_{i(i-1)})$ be the suffix minima of $(\pi_1, \ldots, \pi_{i-1})$, $1 \leqslant i \leqslant n$. Then, the elements $D(i,j)$ of matrix D, $1 \leqslant i, j \leqslant n$, have the following values:

$$D(i,j) = \pi_{ij} \quad \text{if } i > j, \; D(i,i) = \pi_i \text{ and } D(i,j) = 0 \quad \text{if } i < j.$$



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**

Fig. 4. (a) The computation of the seventh row of the d-inversion matrix of the permutation $\pi=(7,9,3,6,2,8,5,4,1)$; (b) The d-inversion matrix D of π, (c) The ds-tree $T[\pi]$.

The computation of the seventh row of the d-inversion matrix D of the sample permutation π is illustrated in Fig. 4(a); the entered d-inversion matrix D is shown in Fig. 4(b).

Let $E(T[\pi])$ be the edge set of the ds-tree $T[\pi]$ rooted at r. Since r inverts every element of π, it follows that $\langle r, D(1,1)\rangle \in E(T[\pi])$. From the d-inversion matrix D we obtain that $\langle r, D(i,i)\rangle \in E(T[\pi])$ if $D(i,1)=\infty$ and $\langle D(i,1), D(i,i)\rangle \in E(T[\pi])$ otherwise, $2 \leqslant i \leqslant n$; see Fig. 4(c).

Sometimes, hereafter, we shall denote by $T[\pi_0]$ the ds-tree $T[\pi]$ rooted at $r=\pi_0$. Thus, the d-inversion matrix of a permutation π contains all the necessary information for constructing the ds-tree $T[\pi_0], T[\pi_1], \ldots, T[\pi_n]$.

We shall call D-`inversion-matrix` the above procedure for the computation of the d-inversion matrix of a permutation π of length n. 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 *over the set* N_n.

Let $\pi=(r,\pi_1,\pi_2,\ldots,\pi_n)$ be a permutation such that $r=n+1$. We have seen that the ds-tree $T[\pi]$ is a tree rooted at r such that: $\{r,\pi_1,\pi_2,\ldots,\pi_n\}$ is its vertex set and $\langle \pi_k,\pi_i\rangle$ is an edge if and only if $\pi_k=\min\{\pi_j \mid \pi_j \in d\text{-}inversion\text{-}set(\pi_i)\}$ $(1 \leqslant i \leqslant n)$. Recall that the ds-tree $T[\pi]$ is also denoted by $T[\pi_0]$, where $r=\pi_0$.

Next, we define the ds-trees $T[\pi_1], T[\pi_2], \ldots, T[\pi_n]$ of a permutation $\pi=(r,\pi_1,\pi_2,\ldots,\pi_n)$. The ds-tree $T[\pi_i]$ $(1 \leqslant i \leqslant n)$ is defined to be a rooted tree such that:
(i) π_i is the root of the tree;
(ii) π_p is a node iff $\pi_i > \pi_p$ and π_p in $(\pi_{i+1},\pi_{i+2},\ldots,\pi_n)$; that is, $i+1 \leqslant p \leqslant n$;
(iii) $\langle \pi_k,\pi_p\rangle$ is an edge iff $\pi_k=\min\{\pi_j \mid \pi_j \in d\text{-}inversion\text{-}set(\pi_p)\}$.
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 the same permutation used in this paper.

Let $T[\pi_i]$ be the ith ds-tree of a permutation π, $1 \leqslant i \leqslant n$. By definition, π_i is the root of the tree and π_j is a node if and only if $\pi_j < \pi_i$ and $i+1 \leqslant j \leqslant n$. We can

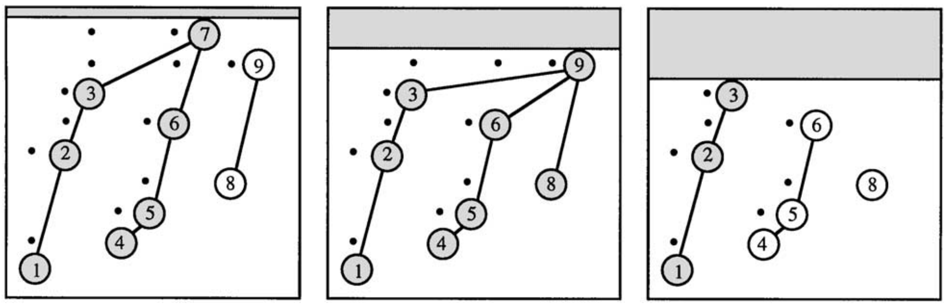


Fig. 5. The ds-trees $T[\pi_1]$, $T[\pi_2]$ and $T[\pi_3]$ of the permutation $\pi=(7,9,3,6,2,8,5,4,1)$; that is, the ds-trees $T[7]$, $T[9]$ and $T[3]$.

therefore compute the edge set of $T[\pi_j]$ using the d-inversion matrix D of π as follows: Set $\langle D(k,i),\pi_k\rangle$ to be an edge of $T[\pi_i]$ if $\pi_k<\pi_i$ and $i+1\leqslant k\leqslant n$.

Thus, the d-inversion matrix of a permutation π contains all the necessary information for constructing the ds-trees $T[\pi_0],T[\pi_1],\ldots,T[\pi_n]$. Sometimes, hereafter, we shall denote by $T[\pi_0]$ the ds-tree $T[\pi]$ rooted at $r=\pi_0$. We construct the ds-tree $T[\pi_i]$ rooted at π_i by computing the parent function, $p(\pi_k)$, for each node π_k such that $\pi_k<\pi_i$ and $i+1\leqslant k\leqslant n$. Next, we list the parallel construction algorithm.

```
Algorithm ds-Trees (Decreasing-Subsequence-Trees):
  Input: A permutation $\pi$ over the set $N_n$;
  Output: The decreasing-subsequence trees $T[\pi_i]$, $0 \leqslant i \leqslant n$;
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 $\pi_1$ to be a child of the
   root $\pi_0$; that is, $p(\pi_1)=\pi_0$;
3. For every $\pi_k$, $2 \leqslant k \leqslant n$, 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 $i$, $1 \leqslant i \leqslant n$, do in parallel
4.1 Set $\pi_i$ to be the root of the tree $T[\pi_i]$;
4.2 For every $\pi_k$, $i+1 \leqslant k \leqslant n$, 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 $T[\pi_0],T[\pi_1],\ldots,T[\pi_9]$ of the permutation $\pi=(7,9,3,6,2,8,5,4,1)$, where $r=\pi_0$ and $\pi_0=10$.

4.2. The link and active-link nodes of a ds-tree

Towards the construction of the cp-tree $T^*[\pi]$, we define for each ds-tree $T[\pi_0],T[\pi_1],\ldots,T[\pi_n]$ two sets of nodes having specific properties. These properties are

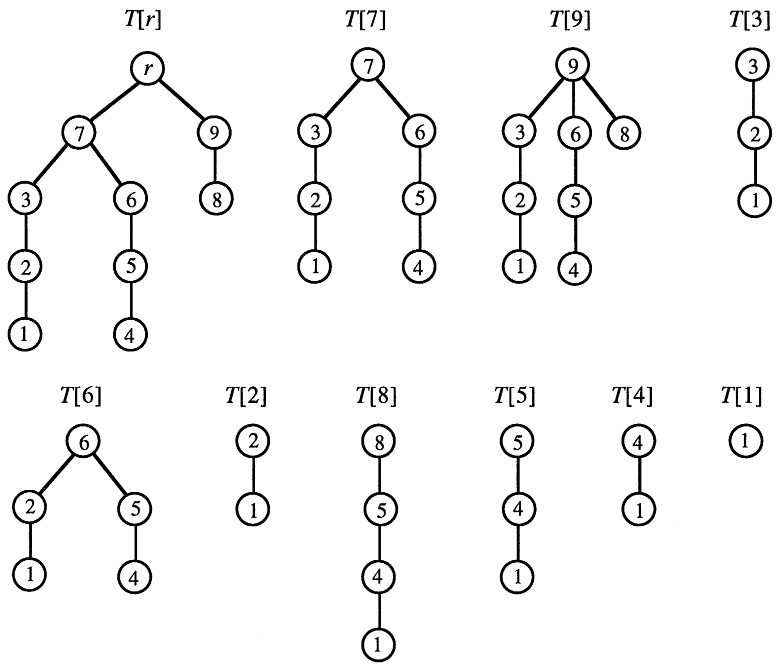


Fig. 6. The ds-trees $T[\pi_0], T[\pi_1], \ldots, T[\pi_9]$ of the permutation $\pi = (7, 9, 3, 6, 2, 8, 5, 4, 1)$. Here, $r = \pi_0 = 10$.

important and play a key role in the cp-tree construction process. For the ds-tree $T[\pi_i]$, $0 \leqslant i \leqslant n$, these two sets are defined as follows:

link-nodes($T[\pi_i]$): It contains all the nodes x of $T[\pi_i]$ having the following property: there exists a node y in $T[\pi_i]$ such that:

(i) $level(y) = level(x)$,

(ii) y is inverted by x.

active-link-nodes($T[\pi_i]$): it contains all the nodes w of *link*-nodes($T[\pi_i]$) having the property: there exists no node x in *link-nodes*($T[\pi_i]$) such that x inverts w.

In this paper, we assume that the sets of link and active-link nodes are ordered sets; that is, the elements in each set are arranged in the same order as they appear in π. For example, the sets of the link and active link nodes of the ds-tree $T[\pi_1]$ of Fig. 7(b) are the following: *link-nodes*($T[\pi_1]$) $= (8, 9, 3)$ and *active-link-nodes*($T[\pi_1]$) $= (8, 9)$.

Before we describe the algorithmic way we can compute the link and active-link nodes of a ds-tree $T[\pi_i]$, $0 \leqslant i \leqslant n$, we show some properties of the link nodes of $T[\pi_i]$ and describe the structure of specific subtrees which contain link nodes. Moreover, we show properties of the link nodes of the trees $T[\pi_i]$ and $T[\pi_j]$, in the case where the root π_j of the tree $T[\pi_j]$ is an active-link node of $T[\pi_i]$, $i < j \leqslant n$.

Let (y,x) be a pair of nodes in the tree $T[\pi_i]$, $0 \leqslant i \leqslant n$. The pair (y,x) is called *link-pair* if y is inverted by x and $level(y) = level(x)$ in $T[\pi_i]$. By definition, the node x in a link-pair (y,x) is a link-node of the tree $T[\pi_i]$.

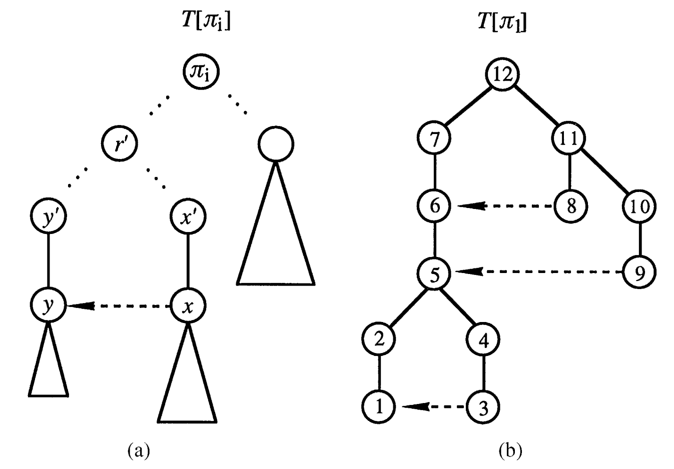


Fig. 7. The link relationship of the link-pair (y,x) and the triplet (y',x',r'); (b) The ds-tree $T[\pi_1]$ of the permutation $\pi=(12,7,11,8,6,10,9,5,2,4,3,1)$.

Let (y,x) be a link-pair in the tree $T[\pi_i]$, $0 \leqslant i \leqslant n$. Let x' and y' be the parents of x and y in $T[\pi_i]$, respectively, and let r' be the lowest common ancestor $lca(y,x)$ of the link-pair (y,x); see Fig. 7(a). Then, the link-pair (y,x) and the triplet (y',x',r') satisfy the following property:

***Link-property* I.**

$$y < y' < x < x' < r',$$

$$\pi^{-1}(r') < \pi^{-1}(y') < \pi^{-1}(x') < \pi^{-1}(x) < \pi^{-1}(y),$$

where, by $\pi^{-1}(x)$ we denote, hereafter, the index of the element x in π; that is $\pi^{-1}(x)=\pi_x^{-1}$ (we use the same notation for the nodes y, y', x' and r').

In Fig. 7(b), the pairs of nodes $(6,8)$, $(5,9)$ and $(1,3)$ are link-pairs in the tree $T[\pi_1]$. It is easy to see that each link-pair satisfies the link property. For example, for the link-pair $(1,3)$ and the triplet $(2,4,5)$ we have: $\pi^{-1}(5) < \pi^{-1}(2) < \pi^{-1}(4) < \pi^{-1}(3) < \pi^{-1}(1)$.

Let (y,x) be a link-pair in the tree $T[\pi_i]$, $0 \leqslant i \leqslant n$. We call *link-subtree* of (y,x) the subtree of $T[\pi_i]$ rooted at $r'=lca(y,x)$; we denote it by *link-subtree*$(\pi_i; y,x)$. In Fig. 7(b), *link-subtree*$(\pi_1; 1,3)$ is the subtree of $T[\pi_1]$ rooted at 5, while *link-subtree*$(\pi_1; 5,9)$ is the subtree of $T[\pi_1]$ rooted at 12. Note that *link-subtree*$(\pi_1; 6,8)$ is also the subtree of $T[\pi_1]$ rooted at 12.

Next, we show some important properties of the link-pairs and the link-subtrees of a tree $T[\pi_i]$, $0 \leqslant i \leqslant n$. These properties are useful for understanding the cp-tree construction algorithm and showing its correctness (see Section 5).

We first show an important property concerning the appearance of the *link-subtree*$(\pi_i; y,x)$ in another tree $T[\pi_j]$, $j \neq i$. Let r be the root of *link-subtree*$(\pi_i; y,x)$. Then, *link-subtree*$(\pi_i; y,x)$ is also a subtree of $T[\pi_j]$ if r is inverted by π_j and $\pi^{-1}(\pi_j)$

$< \pi^{-1}(r)$. It implies that (y,x) is a link-pair in some trees $T[\pi_j]$ with $j < i$ and (y,x) is not a link-pair in any tree $T[\pi_j]$ with $j > i$. In Fig. 7(b), the pair $(1,3)$ is a link-pair in the trees $T[12]$, $T[7]$, $T[11]$, $T[8]$, $T[6]$, $T[10]$ and $T[9]$, while it is not a link-pair in the trees $T[2]$, $T[4]$, $T[3]$ and $T[1]$.

The next property concerns the link-pairs of a tree $T[\pi_i]$. We point out that a tree $T[\pi_i]$ might contain more than one link-pairs $(y_1,x),(y_2,x),\dots,(y_k,x)$ with the same link-node x. For example, let $\pi = (9,3,8,2,5,7,6,4,1)$. It is easy to see that 6 is a link-node and the pairs $(1,6)$ and $(4,6)$ are link-pairs in π. Let r_1 and r_2 be the roots of the subtrees *link-subtree*$(\pi_i; y_1,x)$ and *link-subtree*$(\pi_i; y_2,x)$, respectively. Based on the properties of the lowest common ancestor of a pair of nodes, we can easily show that r_1 (r_2) is an ancestor of r_2 (r_1).

The following algorithmic schemes describe the computations of the node sets *link-nodes*$(T[\pi_i])$ and *active-link-nodes*$(T[\pi_i])$, $0 \leqslant i \leqslant n$.

Computation of the set link-nodes $(T[\pi_i])$, $0 \leqslant i \leqslant n$: Let L_h be an array containing the nodes of the hth level of the tree $T[\pi_i]$ 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 of $T[\pi_i]$.

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

```
for every level $h$ of $T[\pi_i]$, $1 \leqslant i \leqslant n$, do in parallel
  1. 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 $\pi$, where $L_h$ is an array containing the
     nodes of the tree at level $h$.
  2. Compute the prefix maxima $pref\text{-}max\text{-}IL_h$ 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 $IL_h(k) \neq pref\text{-}max\text{-}IL_h(k)$ then $link\text{-}nodes(T[\pi_i]) \leftarrow u$;
     end;
end;
```

We shall refer to the above algorithmic scheme as *Scheme-A*. Let $T[\pi_i]$ be a ds-tree and let *link-nodes*(L_h) be the set of the link nodes of the hth level, where $h \geqslant 1$. Fig. 8 shows the computation of the set *link-nodes*(L_h) using the algorithmic Scheme-A.

Computation of the set active-link-nodes $(T[\pi_i])$, $0 \leqslant i \leqslant n$: By definition, the vertex set *active-link-nodes*$(T[\pi_i])$ contain no pair of 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*$(T[\pi_i])$ that are inverted by an element $u \in$ *link-nodes*$(T[\pi_i])$. This computation can be done using a similar technique as in the computation of the set *link-nodes*$(T[\pi_i])$. Specifically, it can be done using prefix maxima on the elements of the set *link-nodes*$(T[\pi_i])$; We shall

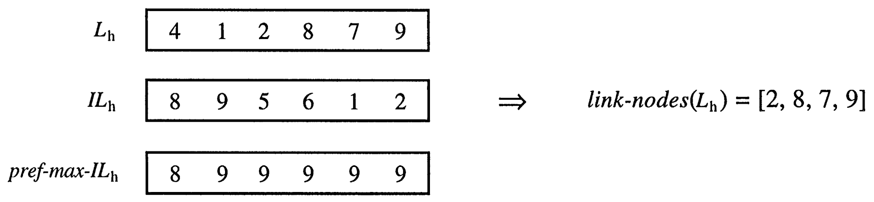


Fig. 8. The computation of the set *link-nodes*(L_h) of a tree $T[\pi_i]$ with h-level nodes $L_h = (4, 1, 2, 8, 7, 9)$.

refer to the algorithmic scheme that computes the active-link nodes of a ds-tree as *Scheme-B*.

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

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

Let us first define some tree operations involved in the cp-tree construction algorithm and give some additional tree notation employed in the rest of the paper.

Let $T[v_1]$ be a tree with n nodes $v_1, v_2, \ldots, v_n$ rooted at v_1 and let $preord(v_i)$ and $info(v_i)$ be the preorder number and the information of the node v_i, respectively. We define the following operations:

(a) *Copy* the tree $T[v_1]$ is defined to be the operation that constructs a tree $T[u_1]$ with n nodes $u_1, u_2, \ldots, u_n$ such that (i) $preord(u_i) = preord(v_i)$, and (ii) $info(u_i) = info(v_i)$, for $i = 1, 2, \ldots, n$. Let $T[v]$ and $T[u]$ be two trees rooted at v and u, respectively, and let u be a node of $T[v]$.

(b) *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 to the root of the tree $T[u]$.

(c) *Remove* a subtree T' of a tree $T[v]$ is defined to be the operation that makes the subtree T' to be an empty tree.

Let $T[\pi_i]$ be the ith ds-tree of a permutation π and let u be an internal node of $T[\pi_i]$, $0 \leqslant i \leqslant n$. By *subtree*$(\pi_i; u)$ we denote the subtree of $T[\pi_i]$ rooted at u.

Let *subtree*$(\pi_i; u)$, *subtree*$(\pi_j; u)$ be subtrees of the ds-trees $T[\pi_i]$ and $T[\pi_j]$, respectively. We say that these two subtrees are *equal*, denoted here as *subtree*$(\pi_i; u) =$ *subtree*$(\pi_j; u)$, if one of the subtrees is a copy of the other.

Let *level-order*(π_i) be a sequence of the nodes of the tree $T[\pi_i]$, which is obtained by visiting the nodes of $T[\pi_i]$ in the level order (or breadth-first order), $0 \leqslant i \leqslant n$. By *level-order*$^{-1}(\pi_i; u)$ we denote the index of the node u in *level-order*(π_i). Note that *level-order*$^{-1}(\pi_i; \pi_i) = 1$. For the sake of consistency, we shall use, hereafter, the notation $level(\pi_i; u)$ to denote the level $level(u)$ of a node u of the tree $T[\pi_i]$; recall that $level(\pi_i; \pi_i) = 0$.

We next give a formal listing of a parallel algorithm for constructing the coloring-permutation tree $T^*[\pi_0]$. The algorithm proceeds as follows:

```
Algorithm cp-Tree (Coloring-Permutation-Tree):
  Input: A permutation $\pi$ over the set $N_n$;
  Output: The coloring-permutation tree $T^*[\pi_0]$;
begin
1. Construct the ds-trees $T[\pi_0], T[\pi_1], \ldots, T[\pi_n]$;
   make all of them to be active trees and paint their nodes white;
2. For every active tree $T[\pi_i]$, $0 \leqslant i \leqslant n$, do in parallel
   2.1 Compute the level $level(\pi_i; u)$ of each node $u$ of $T[\pi_i]$;
   2.2 Compute the set $link\text{-}nodes(T[\pi_i])$;
   2.3 Compute the set $active\text{-}link\text{-}nodes(T[\pi_i])$;
   2.4 If $active\text{-}link\text{-}nodes(T[\pi_i]) = \emptyset$, then make $T[\pi_i]$ to be inactive tree;
3. If $T[\pi_0]$ is an inactive tree, then return($T[\pi_0]$);
4. Compute the set $active\text{-}nodes \leftarrow \bigcup_{0 \leqslant i \leqslant n} active\text{-}link\text{-}nodes(T[\pi_i])$;
5. For every node $u \notin active\text{-}nodes \bigcup \{\pi_0\}$, do in parallel
      Make the ds-tree $T[u]$ to be inactive tree;
6. For every active tree $T[\pi_i]$ and
   every $u \in active\text{-}link\text{-}nodes(T[\pi_i])$, $0 \leqslant i \leqslant n$, do in parallel
   if there exists no $u' \in active\text{-}link\text{-}nodes(T[\pi_j])$ such that $u' = u$ and $j < i$, then
   6.1 Copy the tree $T[u]$, and
       set $copy\text{-}active\text{-}nodes(T[\pi_i]) \leftarrow active\text{-}link\text{-}nodes(T[u])$;
   6.2 Replace the subtree of $T[\pi_i]$ rooted at $u$ with the copy of $T[u]$;
   6.3 Remove the subtree of $T[\pi_i]$ rooted at $y$, if $(y, u)$ is a link-pair;
7. For every active tree $(T[\pi_i])$, $0 \leqslant i \leqslant n$, do in parallel
   7.1 For every node $u \in copy\text{-}active\text{-}nodes(T[\pi_i])$, do in parallel
       Find all the nodes $u_1, u_2, \ldots, u_p$ of $T[\pi_i]$ such that $u_1 = u_2 = \cdots = u_p = u$ and
       $level\text{-}order^{-1}(\pi_i; u_1) < level\text{-}order^{-1}(\pi_i; u_2) < \cdots < level\text{-}order^{-1}(\pi_i; u_p)$;
       Make the children of the node $u_i$, $1 \leqslant i \leqslant p-1$, to be children of the node
       $u_p$;
   7.2 For every node $u \in T[\pi_i]$, do in parallel
       Paint the node $u$ black if there exists a node $u'$ in $T[\pi_i]$ such that
       $u = u'$ and $level\text{-}order^{-1}(\pi_i; u) < level\text{-}order^{-1}(\pi_i; u')$;
   7.3 Remove the black nodes of the tree $T[\pi_i]$; note that, the root of the
       tree $T[\pi_i]$ is always a white node;
8. Make the sets of link-nodes, active-link-nodes and copy-active-nodes to be
      empty sets for every active tree $T[\pi_i]$, $0 \leqslant i \leqslant n$, go to step 2;
end;
```

Before we prove the correctness of the algorithm, let us comment on Steps 6 and 7.

Remark 4.1. In Step 6 some active trees $T[u]$ are copied and moved in the tree $T[\pi_i]$ (Steps 6.1 and 6.2) and some subtrees of the tree $T[\pi_i]$ are removed from it (Step

6.3). Since the levels of each node x of $T[\pi_i]$ and each node y of $T[u]$ are known (Step 2), it follows that the new level of the node y in the tree $T[\pi_i]$ is also known. Thus, there is no need for computing the level order of the tree $T[\pi_i]$ before the execution of Step 7.

Remark 4.2. From the properties of the link and active-link nodes shown in Section 4.2, we have that a tree $T[\pi_i]$ might contain more than one link-pairs $(y_1,u),(y_2,u),\ldots,(y_k,u)$ with the same active-link node u. Moreover, it is also possible to have two link-pairs (y_1,u_1) and (y_2,u_2) in $T[\pi_i]$ such that y_1 is an ancestor of y_2 and the nodes u_1,u_2 are active-link nodes. In both cases, after executing Step 6, a subtree of $T[\pi_i]$ might appear twice in $T[\pi_i]$; see *link-subtree*$(\pi_1;1,3)$ in Fig. 7(b). In this example, the subtree *link-subtree*$(\pi_1;1,3)$ appears twice in $T[\pi_i]$ since neither node 8 nor node 9 is an active-link node in a tree $T[i]$, where $\pi^{-1}(i) < \pi^{-1}(12)$; that is, both active trees $T[8]$ and $T[9]$ are copied and moved in the active tree $T[12]$. Step 7 removes all the duplicate nodes from a tree $T[\pi_i]$ using the level order of the nodes of $T[\pi_i]$.

Remark 4.3. The node set *copy-active-nodes*$(T[\pi_i])$, which is computed in Step 6.1, contains the active nodes of all the active trees $T[u]$ which are copied and moved in the tree $T[\pi_i]$, $0 \leqslant i \leqslant n$, during an iteration of Steps 2–7 of the algorithm. In the example of Fig. 7(b), *copy-active-nodes*$(T[12]) = (3)$ (we assume that the set of the copy-active nodes is an ordered set). Let $u_1,u_2,\ldots,u_p$ be the active-link nodes of the tree $T[\pi_i]$ during an iteraction of Steps 2–7. It is easy to see that, if no active tree $T[u_1],T[u_2],\ldots,T[u_p]$ is copied and moved in $T[\pi_i]$ (Step 6), then the tree $T[\pi_i]$ contains the same active nodes $u_1,u_2,\ldots,u_p$ in the next iteration. On the other hand if all the active trees $T[u_1],T[u_2],\ldots,T[u_p]$ are copied and moved in $T[\pi_i]$ during an iteration, then *active-link-nodes*$(T[\pi_j]) \subseteq$ *copy-active-nodes*$(T[\pi_i])$ in the next iteration; see Lemma 5.4. □

Remark 4.4. It is easy to see that Step 6.3 can be replaced by the following statement: "Paint the nodes of the *subtree*$(\pi_i;y)$ of the tree $T[\pi_i]$ black if (y,u) is a link-pair in $T[\pi_i]$". In this case, Step 7.3 removes all the nodes of the subtree *subtree*$(\pi_i;y)$.

5. Correctness

In this section we prove the correctness of algorithm `cp-Tree`. We first show some properties of the ds-trees $T[\pi_0],T[\pi_1],\ldots,T[\pi_n]$ of a permutation π over the set N_n. Recall that, throughout the paper, we use the notation $\pi^{-1}(i)$ to denote the index of the element i in π; that is, $\pi_i^{-1} = \pi^{-1}(i)$.

Lemma 5.1. *Let $T[\pi_i]$ be the ith ds-tree of a permutation π, and let z be a link-node of $T[\pi_i]$ at level h. Let y be a node at level $h-k$ such that z inverts y, $0 \leqslant k \leqslant h-1$.*

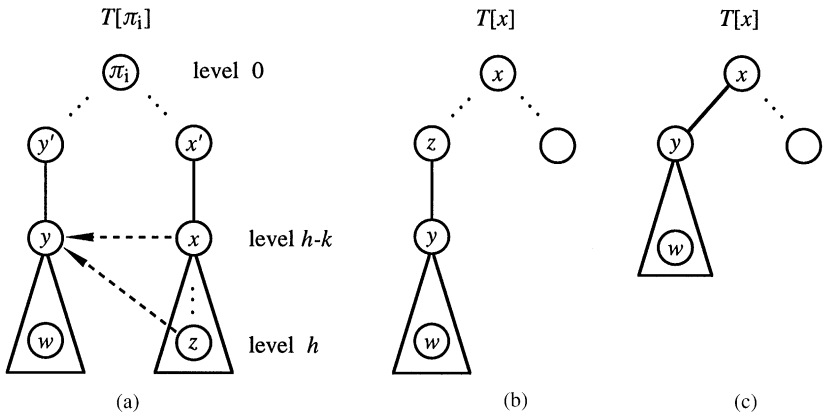


Fig. 9. (a) The tree $T[\pi_i]$ and the inversion relationship of the nodes z, x and y; (b) The tree $T[x]$ in the case where there exists a node z in *subtree*$(\pi_i;x)$ such that z inverts y; (c) The tree $T[x]$ in the case where either there exists no node z in *subtree*$(\pi_i;x)$ such that z inverts y or *subtree*$(\pi_i;x)$ is empty.

Then, there exists a node x in the path from the root π_i to node z having the following properties:
(i) $level(\pi_i;x) = level(\pi_i;y)$.
(ii) *x inverts y.*

Proof. Let z, y be nodes of the tree $T[\pi_i]$ such that $level(\pi_i;z) = h$, $level(\pi_i;y) = h-k$ and z inverts y. Let $P = (\pi_i, \dots, z)$ be the path from the root π_i to node z. Then, there exists a node x in P such that $level(\pi_i;x) = h-k$; see Fig. 9(a). Since z inverts y, it follows that y is not a node of P, $z > y$ and $\pi^{-1}(z) < \pi^{-1}(y)$. Moreover, $x > z$ and $\pi^{-1}(x) < \pi^{-1}(z)$. Hence, $x > y$ and $\pi^{-1}(x) < \pi^{-1}(y)$. Thus, x inverts y. □

Lemma 5.2. *Let z be a link-node of the tree $T[\pi_i]$ at level h and let y be a node at level $h-k$ such that z inverts y, $0 \leqslant k \leqslant h-1$. Let x be the ancestor of z at level $h-k$ that inverts y. Then, subtree$(\pi_i;y)$ = subtree$(x;y)$.*

Proof. Since x inverts y, it follows that $x > y$ and $\pi^{-1}(x) < \pi^{-1}(y)$. Moreover, the parent y' of the node y is such that $y' > y$ and $\pi^{-1}(y') < \pi^{-1}(x) < \pi^{-1}(y)$. It is easy to see that $y' < x$; specifically, y' is the smallest element in the range $\pi^{-1}(1) \cdots \pi^{-1}(x)$ that is larger than y. Let w be an arbitrary node of the *subtree*$(\pi_i;y)$. Obviously, $w < y$ and $\pi^{-1}(w) > \pi^{-1}(y)$. We shall prove that w is also a node of the *subtree*$(x;y)$. Suppose the contrary. Then, there exists a node y'' in the range $\pi^{-1}(x) \cdots \pi^{-1}(y)$, such that $y'' < y$ and $y'' > w$. In this case, both nodes y'' and y have the same parent y' in the tree $T[\pi_i]$. Since $y'' > w$, it follows that y'' is an ancestor w. Thus, w is not a node of the *subtree*$(\pi_i;y)$; a contradiction. □

Lemma 5.3. *Let z be a link-node of the tree $T[\pi_i]$ at level h and let y be a node at level $h-k$ such that z inverts y, $0 \leqslant k \leqslant h-1$. Let x be the ancestor of z at level*

$h-k$ that inverts y. Then, the following properties hold:
(i) *if $w \in subtree(\pi_i; y)$ then $w \in T[x]$*;
(ii) *$level(\pi_i; w) \leqslant level(\pi_i; x) + level(x; w)$.*

Proof. It follows immediately from Lemma 5.2. □

Lemma 5.4. *Let $x \in$ active-link-nodes$(T[\pi_i])$ and $w \in$ active-link-nodes$(T[x])$. If there exists no node $x' \in$ active-link-nodes$(T[\pi_j])$ such that $x' = x$ and $j < i$, then w becomes an active-link node of $T[\pi_i]$ after executing Step 7 of algorithm* cp-Tree.

Proof. It follows immediately from the operations performed in Steps 6 and 7 of the algorithm. □

Lemma 5.5. *Let w be a node of a tree $T[\pi_i]$, $0 \leqslant i \leqslant n$, and let $w \notin$ active-nodes after the kth iteration of Step 7 of algorithm* cp-Tree. *Then, $w \notin$ active-nodes after the $(k+1)$th iteration.*

Proof. Suppose that $w \in$ *active-nodes* after the $(k+1)$th iteration of Step 7, and let $w \in$ *active-link-nodes*$(T[\pi_i])$, $0 \leqslant i \leqslant n$. Since w is a link node of $T[\pi_i]$, it follows that there exists a node u in $T[\pi_i]$ such that u is inverted by w and $level(\pi_i; u) = level(\pi_i; w)$; that is, (u, w) is a link-pair in $T[\pi_i]$. Let w' and u' be the parents of w and u in the tree $T[\pi_i]$, respectively, and let s' be the lowest common ancestor of the nodes w and u; that is, s' is the root of the link-subtree *link-subtree*$(\pi_i; u, w)$; see Fig. 10(a). Since $w \notin$ *active-nodes* after the kth iteration of Step 7, we have that $w \notin$ *active-link-nodes*$(T[\pi_i])$. We distinguish two alternatives; see Fig. 10(b).

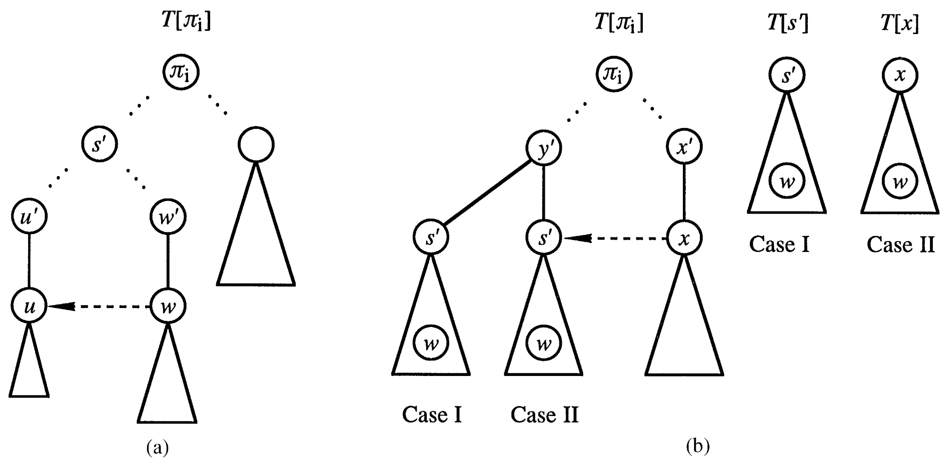


Fig. 10. (a) The structure of the *link-subtree*$(\pi_i; u, w)$; (b) The link relationship of the *link-subtree*$(\pi_i; u, w)$ and link-pair (y, x).

Case I: There exists no node $x \in$ *active-link-nodes*$(T[\pi_i])$ such that x inverts s'. Since $w \in$ *active-link-nodes*$(T[\pi_i])$ and *link-subtree*$(\pi_i; u, w)$ is the link-subtree rooted at s', it follows that w is an active-link node in $T[s']$. Thus, $w \in$ *active-nodes*; a contradiction.

Case II: There exists a node $x \in$ *active-link-nodes*$(T[\pi_i])$ such that x inverts s'. Without loss of generality, we suppose that $level(\pi_i; s') = level(\pi_i; x)$; otherwise, there exists a node x' in the path from the root π_i to node x having this property (see Lemma 5.1). Thus, (s', x) is a link-pair in $T[\pi_i]$. It follows that, the link-subtree *link-subtree*$(\pi_i; u, w)$ is also a link-subtree of $T[x]$ and, therefore, w is an active-link node in $T[x]$. Thus, $w \in$ *active-nodes*; a contradiction. □

Remark 5.1. The main operations performed by Step 7 of Algorithm `cp-Tree` are (i) the movement of each subtree *subtree*$(\pi_i; u_i)$ of the active tree $T[\pi_i]$, where $u_i \in$ *active-link-nodes*$(T[\pi_i])$, so that each node u of the *subtree*$(\pi_i; u_i)$ has the greater possible distance from the root π_i, and (ii) the removal of all the duplicate nodes from the active tree $T[\pi_i]$, $0 \leqslant i \leqslant n$. Recall that, we can efficiently handle the removal operation of subtrees of Step 6.3 using Step 7.3 (see Remark 4.4). Based on these two operations of Step 7 and the copy, replace and remove operations of Step 6, it is easy to see that Lemmas 5.4 and 5.5 also hold at the end of each execution of Step 6. □

Let x, y, z be nodes of a ds-tree. Hereafter, x *inverts* y *inverts* $z = (x$ *inverts* $y)$ and $(y$ *inverts* $z)$. Fig. 11 shows a ds-tree $T[\pi_i]$, in which the link nodes $x_1, x_2, \ldots, x_k$ have the property that x_1 *inverts* x_2 *inverts* … *inverts* x_k. In this case we say that the tree $T[\pi_i]$ 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 Steps 2–8 of algorithm `cp-Tree`.

Suppose that the ds-trees $T[\pi_0], T[\pi_1], \ldots, T[\pi_n]$ of a permutation π have *single* link nodes. Based on the results of Lemmas 5.1–5.5, it is easy to see that there exists a sequence $T_0 = (T[\pi_0], T[\pi_{p(1)}], T[\pi_{p(2)}], \ldots, T[\pi_{p(k)}])$ of length k having the following property:

***Link-property* II.**

$$\pi_{p(1)} \in \textit{active-link-nodes}\ (T[\pi_0]), \text{ and}$$

$$\pi_{p(i)} \in \textit{active-link-nodes}\ (T[\pi_{p(i-1)}]), \quad i = 2, 3, \ldots, k.$$

Consider now the following algorithmic scheme:

```
for $i = 1, 2, \ldots, k$ do
 1. Copy the tree $T[\pi_{p(i)}]$;
 2. Replace the subtree of $T[\pi_0]$ rooted at $\pi_{p(i)}$ with the copy of $T[\pi_{p(i)}]$;
 3. Remove the subtree $subtree(\pi_0; y)$, where $y$ is a node such that $\pi_{p(i)}$ inverts $y$ and
    $level(\pi_0; y) = level(\pi_0; \pi_{p(i)})$;
end;
```

The above algorithmic scheme constructs the tree $T^*[\pi_0]$. We shall refer to this scheme as *Scheme-C*; It is easy to see that the parallel implementation of Scheme-C

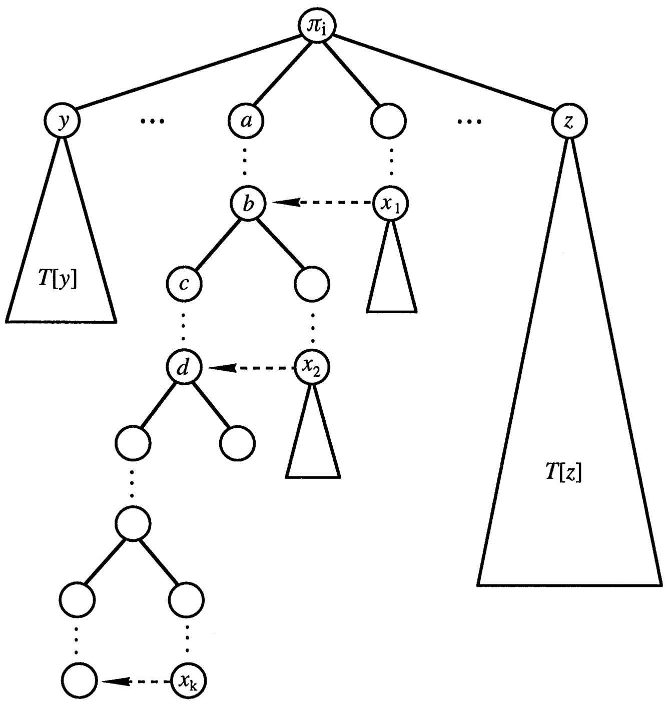


Fig. 11. The structure of the link-nodes of a ds-tree $T[\pi_i]$ of a permutation π, having single link-nodes. Subtrees $T[y]$ and $T[z]$ have no link-nodes.

corresponds to Step 6 of algorithm cp-Tree. Note that if a tree $T[\pi_i]$ has single link nodes, then it contains no black nodes after executing Step 7.2.

The next lemma summarizes two important properties of the active-link nodes of the ds-trees $T[\pi_0], T[\pi_1], \ldots, T[\pi_n]$ in the case where all the ds-trees have single link nodes. These properties, along with the property provided by Lemma 5.5, allow us to estimate the total number of iterations of Steps 2–8 of algorithm cp-Tree.

Lemma 5.6. *Let x be an active-link node of the tree $T[\pi_i]$ and let w be an active-link node of the tree $T[x]$. Let $level(\pi_i; x) = level(x; w) = h$. Then, after executing Step 7 of algorithm* cp-Tree, *we have*:

(i) $x \notin$ *active-link-nodes*$(T[\pi_i])$;
(ii) $w \in$ *active-link-nodes*$(T[\pi_i])$ and $level(\pi_i; w) = 2h$.

Proof. It follows immediately from Lemmas 5.3 and 5.4, and the tree operations performed in Steps 6 and 7 of algorithm cp-Tree. □

Lemma 5.6 says that each active-link node of an active tree $T[\pi_i]$ duplicates its level (or, equivalently, halves its height) after each execution of Step 7 of algorithm cp-Tree. Moreover, by Lemma 5.5 we have that if a node x is not an active-link node

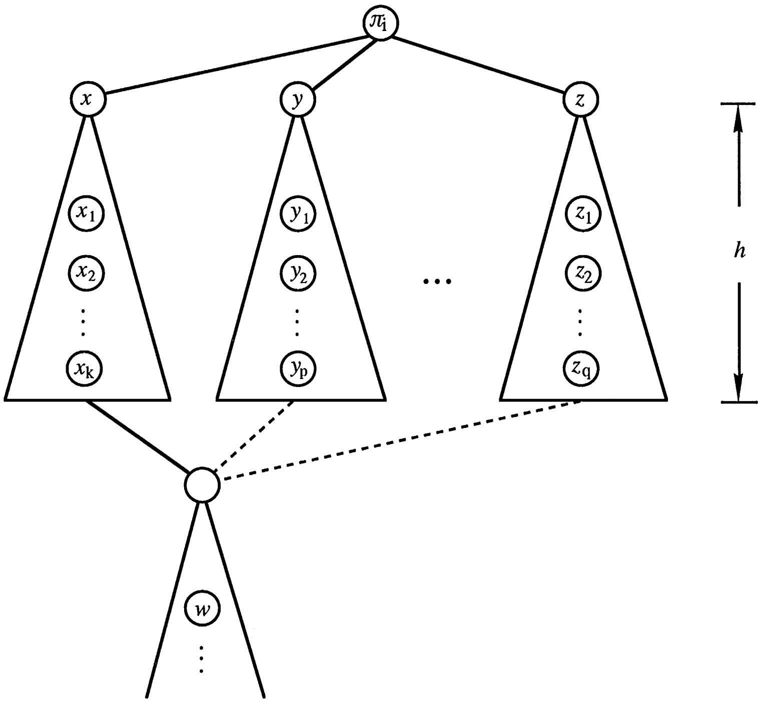


Fig. 12. The structure of the link-nodes of a ds-tree $T[\pi_i]$ of a permutation π having multiple link nodes.

at the end of an iteration of Step 7, then it is not an active-link node at the end of the next iteration of Step 7. Note that these results also hold at the end of Step 6; see Remark 5.1.

From the above discussion and Lemma 5.6, we conclude that the tree $T[\pi_0]$ contains no active-link node after executing $O(\log k)$ times Steps 2 through 8 of algorithm `cp-Tree`, where k is the length of the sequence $T_0 = (T[\pi_0], T[\pi_{p(1)}], T[\pi_{p(2)}], \dots, T[\pi_{p(k)}])$. Thus, we have the following result.

Lemma 5.7. *If the ds-trees* $T[\pi_0], T[\pi_1], \dots, T[\pi_n]$ *of a permutation* π *over the set* N_n *have single link nodes, then the cp-tree* $T^*[\pi]$ *is computed after* $O(\log n)$ *iterations of Steps* 2–8 *of algorithm* `cp-Tree`.

Let $T[\pi_i]$ be a ds-tree, and let *subtree*$(\pi_i; y)$ and *subtree*$(\pi_i; z)$ be two subtrees such that $y < z$. It is also possible that (i) no node in *subtree*$(\pi_i; z)$ inverts a node in *subtree*$(\pi_i; y)$, and that (ii) there exist link nodes $y_1 \in$ *subtree*$(\pi_i; y)$ and $z_1 \in$ *subtree* $(\pi_i; z)$ such that no node in $T[\pi_i]$ inverts nodes y_1 and z_1. Then, y_1 and z_1 are two active-link nodes in the tree $T[\pi_i]$. In this case we say that $T[\pi_i]$ has *multiple* link nodes. The structure of a ds-tree having multiple link nodes is shown in Fig. 12.

Let us now compute the total number of iterations of Steps 2–8 of algorithm `cp-Tree` for the construction of the cp-tree $T^*[\pi]$ in the case where the ds-trees

$T[\pi_0], T[\pi_1], \ldots, T[\pi_n]$ of the given permutation have multiple link nodes. Let

$$\begin{aligned}
L_0 &= (\pi_{01}),\\
L_1 &= (\pi_{11}, \pi_{12}, \ldots, \pi_{1b_1}),\\
L_2 &= (\pi_{21}, \pi_{22}, \ldots, \pi_{2b_2}),\\
&\vdots\\
L_{\mathrm{k}} &= (\pi_{k1}, \pi_{k2}, \ldots, \pi_{kb_k})
\end{aligned}$$

be $k+1$ ordered node sets which contain the active-link nodes of all the active trees during an iteration. Set L_1 contains the active-link nodes of the active tree $T[\pi_{01}]$, where $\pi_{01} = \pi_0$, set L_2 contains the active-link nodes $\pi_{21}, \pi_{22}, \ldots, \pi_{2b_2}$ of the active trees $T[\pi_1], T[\pi_{12}], \ldots, T[\pi_{1b_2}]$ such that π_{2i} is not a link node in $T[\pi_{1j}]$, where $i \neq j$ and $1 \leqslant i, j \leqslant b_2$, and so forth. In fact, the sequences $L_1, L_2, \ldots, L_k$ are the layers of the active-link nodes of the tree $T[\pi_{01}]$. Thus, it follows that $\pi^{-1}(\pi_{ij}) < \pi^{-1}(\pi_{1p})$, for $i \leqslant l$. Moreover, $b_1 + b_2 + \cdots + b_k < n$ (see, link-property II) and the active tree $T[\pi_{ij}]$ contains no more than $n + 1 - (b_1 + b_2 + \cdots + b_{i-1})$ nodes, $1 \leqslant i \leqslant k$. We consider the following two alternatives: *Multiple case* I, and *Multiple case* II.

Multiple case I: The sets of active-link nodes of the active trees $T[\pi_{i1}], T[\pi_{i2}], \ldots, T[\pi_{ib_i}]$ $(1 \leqslant i \leqslant k-1)$ are pairwise disjoint.

Let $T[\pi_i]$ be an active tree and let $\pi_{i1}, \pi_{i2}, \ldots, \pi_{ip}$ be the active-link nodes $T[\pi_i]$, $1 \leqslant p \leqslant b_i$. Since the active trees $T[\pi_{i1}], T[\pi_{i2}], \ldots, T[\pi_{ip}]$ contain distinct active-link nodes, it is easy to see that any operation performed in Steps 6 and 7 of the algorithm on the nodes of *subtree*$(\pi_i; \pi_{ij}), 1 \leqslant j \leqslant p$, does not affect the link-node relationship of the nodes of any other subtree *subtree*$(\pi_i; \pi_{ij'})$, where $j \neq j'$ and $1 \leqslant j, j' \leqslant p$.

Based on the results of this section, we can easily conclude that the multiple case I is identical, with respect to the link-node relationship, to the case where all the active ds-trees have single link nodes. Thus, after executing $O(\log n)$ times Steps 2–8 of algorithm `cp-Tree` there exists no tree $T[\pi_i]$ with active-link nodes, $0 \leqslant i \leqslant n$; that is, $T[\pi_0] = T^*[\pi]$.

Multiple case II: There exists a node that is active-link node in more than one of the active trees $T[\pi_{i1}], T[\pi_{i2}], \ldots, T[\pi_{ib_i}]$ $(1 \leqslant i \leqslant k-1)$.

We consider the worst-case scenario where each of the active trees $T[\pi_{i1}], T[\pi_{i2}], \ldots, T[\pi_{ib_i}]$ contains all the b_{i+1} active-link nodes of the set L_{i+1} $(1 \leqslant i \leqslant k-1)$.

Let $T[\pi_i]$ be an active tree and let $\pi_{i1}, \pi_{i2}, \ldots, \pi_{ib_i}$ be the active-link nodes of $T[\pi_i]$ during the ith iteration of Steps 2–8 of the algorithm. Let h be the length of the minimum path (number of edges) from the root π_i to an active-link node π_{ij}; that is, *level*$(\pi_i; \pi_{ij}) \geqslant h$, for every j, where $1 \leqslant j \leqslant b_i$. Since $\pi_{i1}, \pi_{i2}, \ldots, \pi_{ib_i}$ are active-link nodes of the tree $T[\pi_i]$, it follows that the ds-trees $T[\pi_{i1}], T[\pi_{i2}], \ldots, T[\pi_{ib_i}]$ are active during the ith iteration. Let $\pi_{j1}, \pi_{j2}, \ldots, \pi_{jb_j}$ be the active-link nodes of the first ds-tree $T[\pi_{i1}]$. From the structure of the worst-case scenario we have that, during the ith iteration, the length of the minimum path from the root π_{i1} of the ds-tree $T[\pi_{i1}]$ to an active-link node π_{jp}, $1 \leqslant p \leqslant b_j$, is equal to h; that is, *level*$(\pi_{i1}; \pi_{jp}) \geqslant h$, for every

p, where $1 \leqslant p \leqslant b_j$. On the other hand, during the same iteration, for the other trees $T[\pi_{i2}], \ldots, T[\pi_{ib_i}]$ we have that the nodes $\pi_{j1}, \pi_{j2}, \ldots, \pi_{jb_j}$ are link nodes in all these trees $T[\pi_{i2}], \ldots, T[\pi_{ib_i}]$ and $level(\pi_{it}; \pi_{jp}) \leqslant h$, where $2 \leqslant t \leqslant b_i$ and $1 \leqslant p \leqslant b_j$.

We consider now the $(i+1)$th iteration of Steps 2–8. In this iteration, the active trees $T[\pi_{i1}], T[\pi_{i2}], \ldots, T[\pi_{ib_i}]$ are copied and moved in the tree $T[\pi_i]$. We have that (i) the active-link nodes $\pi_{j1}, \pi_{j2}, \ldots, \pi_{jb_j}$ of the tree $T[\pi_{i1}]$ have level greater than or equal to h in $T[\pi_{i1}]$, (ii) the active-link nodes of the tree $T[\pi_{i1}]$ are link nodes in each tree $T[\pi_{i2}], \ldots, T[\pi_{ib_i}]$, and (iii) each of the active-link nodes $\pi_{s1}, \pi_{s2}, \ldots, \pi_{sb_s}$ of the tree $T[\pi_{it}]$, $2 \leqslant t \leqslant b_i$, that is not an active-link node of the tree $T[\pi_{i1}]$, is an ancestor of some of the active-link nodes $\pi_{j1}, \pi_{j2} \ldots, \pi_{jb_j}$ of the tree $T[\pi_{i1}]$. Since Step 7.1 performs only link operations on the nodes of the tree $T[\pi_i]$ which are active-link nodes in the trees $T[\pi_{i1}], T[\pi_{i2}], \ldots, T[\pi_{ib_i}]$ (in fact, Step 7.1 change the parent relation of the nodes of $T[\pi_i]$ according to max level-order), it follows that after executing Steps 7.2 and 7.3 the active-link nodes of the tree $T[\pi_i]$, if there exist such nodes, have level greater than or equal to $2h$. Thus, after executing $O(\log n)$ times Steps 2–8 of algorithm `cp-Tree` there exists no tree $T[\pi_i]$ with active-link nodes, $0 \leqslant i \leqslant n$; that is, $T[\pi_0] = T^*[\pi]$.

From Lemma 5.7 and the cases Multiple cases I and II we have that after executing $O(\log n)$ times Steps 2–8 of algorithm `cp-Tree` there exists no tree $T[\pi_i]$ with active-link nodes, $0 \leqslant i \leqslant n$. Thus, the results of this section can be summarized in the following theorem.

Theorem 5.1. *Given a permutation π over the set N_n, algorithm* `cp-Tree` *constructs the cp-tree $T^*[\pi]$ after $O(\log n)$ iterations of Steps 2–8.*

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). 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 complexity. Moreover, in this model, the computation of the postorder, preorder, inorder and level order of a tree (binary tree for inorder), 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 on the EREW PRAM model within the same time-processor complexity); see [8,15].

6.1. The D-inversion-matrix algorithm

We first show the time and processor complexity of the algorithm for the computation of the d-inversion matrix of a permutation π. It is well-known that the suffix minima of n elements can be computed in $\mathrm{O}(\log n)$ time using $\mathrm{O}(n/\log n)$ processors on the EREW PRAM model; note that the suffix minima problem is based on the computation of the prefix-sums of n elements [8,15]. Thus, the following theorem holds.

Theorem 6.1. *Let π be a permutation over the set N_n. The d-inversion matrix of π can be computed in $\mathrm{O}(\log n)$ time using $\mathrm{O}(n^2/\log n)$ processors on the CREW PRAM model.*

6.2. The ds-trees algorithm

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

The algorithm consists of four 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 $\mathrm{O}(\log n)$ time using $\mathrm{O}(n^2/\log n)$ processors.

Step 2: Obviously, it takes $\mathrm{O}(1)$ sequential time.

Step 3: Having computed the d-inversion matrix, this step requires $\mathrm{O}(1)$ time and $\mathrm{O}(n)$ processors or $\mathrm{O}(\log n)$ time and $\mathrm{O}(n/\log n)$ processors.

Step 4: The time and processor requirement of Steps 4.1 and 4.2 are the same as that required for Steps 2 and 3, respectively. Both substeps are executed for every i, $1 \leqslant i \leqslant n$, in parallel. Therefore, Step 4 requires $\mathrm{O}(1)$ time and $\mathrm{O}(n^2)$ processors or $\mathrm{O}(\log n)$ time and $\mathrm{O}(n^2/\log n)$ processors.

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

Theorem 6.2. *Given a permutation π over the set N_n, the ds-trees of π can be constructed in $\mathrm{O}(\log n)$ time using $\mathrm{O}(n^2/\log n)$ processors on the CREW PRAM model.*

6.3. The cp-tree algorithm

Next, we compute the time and processor complexity of 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 by the algorithm.

We have mentioned that the level of each node of a tree can be computed in $\mathrm{O}(\log n)$ time using $\mathrm{O}(n/\log n)$ processors on the EREW PRAM model [8,15]. We next compute the complexity for the computation of 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 $\mathrm{O}(\log n)$ time using $\mathrm{O}(n_h/\log n)$ 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 $\mathrm{O}(\log n)$ time using $\mathrm{O}(n_h/\log n)$ processors on the EREW PRAM model [8,15]. Therefore, the execution of Scheme-A requires $O(\log n)$ time and $O(n/\log n)$ processors on the EREW PRAM model.

In a similar way we can show that Scheme-B, which computes the active-link nodes of a ds-tree, is executed in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM model. Thus, the following lemma holds.

Lemma 6.1. *The sets of link and active-link 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.*

The preceding lemma gives us the time and processor complexity of Step 2 of algorithm `cp-Tree`. We focus now on Steps 6 and 7, which are the crucial steps for the processor complexity of the algorithm. Obviously, an active tree $T[u]$ with n nodes can be copied in $O(\log n)$ time using a total of $O(n/\log n)$ processors on the EREW PRAM model. The copy tree $T[u]$ is moved in an active tree $T[\pi_i]$ by simply making the copy tree $T[u]$ subtree of the tree $T[\pi_i]$; in fact, we make the parent $p(u)$ of the node u in $T[\pi_i]$ to be the parent of the root u of the copy tree $T[u]$. This operation takes $O(1)$ sequential time. Next, we prove the following lemmas.

Lemma 6.2. *Each iteration of Steps* 6 *of algorithm* `cp-Tree` takes $O(\log n)$ *time and requires* $O(n^2/\log n)$ *processors on the CREW PRAM model.*

Proof. Let $T[\pi_0], T[\pi_1], \ldots, T[\pi_n]$ be the ds-trees of a permutation π over the set N_n. Suppose that in each iteration of the algorithm some active trees contain single link nodes and some trees contain multiple link nodes. In any case, we have that no more that n active trees are copied and moved in some other active trees. Since the active trees $T[x_{ij}]$ ($1 \leqslant i \leqslant k$ and $1 \leqslant j \leqslant b_i$) contain no more than $n+1-(b_1+b_2+\cdots+b_{i-1})$ nodes, it follows that $O(n^2)$ nodes are copied and moved during an iteration of Steps 6.1 and 6.2.

Using standard parallel algorithmic techniques, we can copy and move $O(n^2)$ nodes in $O(\log n)$ time with $O(n^2/\log n)$ processors on the CREW PRAM model.

We next compute the time and processor complexity for the removal of all the subtrees *subtree*$(\pi_i; y)$ of an active tree $T[\pi_i]$ which contain a link-pair (y, w), where w is an active node, $0 \leqslant i \leqslant n$; this operation is performed in Step 6.3. To this end, we need to determine for each active-link node w of $T[\pi_i]$ all the nodes y of $T[\pi_i]$ such that (y, w) is a link-pair, $0 \leqslant i \leqslant n$. This computation can be done in $O(\log n)$ time with $O(n/\log n)$ processors using the well-known interval broadcasting and array packing parallel techniques on the array *level-order*(π_i). Recall that the node sequence *level-order*(π_i) is obtained by visiting the nodes of $T[\pi_i]$ in the level order; in a parallel setting, it is computed using the Euler-tour technique [8,15]. Since the remove operation is executed in $O(1)$ sequential time and the interval broadcasting, array packing and Euler-tour techniques do not require simultaneous memory access, we conclude that Step 6.3 is executed in $O(\log n)$ time with $O(n^2/\log n)$ processors on the EREW PRAM model. Thus, the whole step is executed in $O(\log n)$ time with $O(n^2/\log n)$ processors on the CREW PRAM model, and, thus, the lemma holds. □

Lemma 6.3. *Each iteration of Steps* 7 *of algorithm* `cp-Tree` *takes* $O(\log n)$ *time and requires* $O(n^2/\log n)$ *processors on the CREW PRAM model.*

Proof. We consider now the operations performed in Step 7 of the algorithm. Let us first focus on Step 7.1 and show how we can find for every node u of the set *copy-active-nodes*$(T[\pi_i])$, $0 \leqslant i \leqslant n$, all the nodes $u_1, u_2, \ldots, u_p$ of the tree $T[\pi_i]$ such that $u_1 = u_2 = \cdots = u_p = u$, and, then, find among them the node w with the *max* index in the order *level-order*(π_i).

We have shown that after executing Step 6 the total number of nodes of the active trees is in $O(n^2)$; see Lemma 6.2. Let $\pi_{i1}, \pi_{i2}, \ldots, \pi_{ib_i}$ be the active-link nodes of the active tree $T[\pi_i]$, $0 \leqslant i \leqslant n$. For the active tree $T[\pi_i]$ we define the matrix $B_i(0..b_i, 1..n)$, where b_i is the number of the active-link nodes in $T[\pi_i]$, and set $B_i(\pi_{ij}, u)$:=*level-order*$^{-1}(\pi_i; u)$ if u is a node of the copy tree $T[\pi_{ij}]$, $1 \leqslant j \leqslant b_i$. This operation needs $O(\log n)$ time and $O(b_\mathrm{i}(n/\log n))$ processors on the CREW PRAM, since the level order of a tree is computed in $O(\log n)$ time with $O(n/\log n)$ processors on the same model of computation; see [8,15].

Let $u_1, u_2, \ldots, u_p$ denote the node u of the corresponding copy trees, say, $T[\pi_{i1}]$, $T[\pi_{i2}], \ldots, T[\pi_{ip}]$, which are moved in the tree $T[\pi_i]$, $0 \leqslant p \leqslant b_i$. Note that, $u \in$ *active-link-nodes*$(T[\pi_i])$. Let u_m be the node u with the *max* index in the order *level-order*(π_i) among the nodes $u_1, u_2, \ldots, u_p$. The computation of finding the node u_m can be done in $O(\log n)$ sequential time if $b_i = c \log n$ for same constant c, or in $O(\log b_i)$ parallel time with $O(b_i/\log b_i)$ processors if $b_i = n^{c_i}$ for $0 < c_i < 1$. After finding the node u_m, we make the nodes $u_1, \ldots, u_{m-1}, u_{m+1}, \ldots, u_p$ to be children of the node u_m in $O(1)$ time using $O(b_i)$ processors. Note that, in the description of Step 7.1 of the algorithm, we make the children of the nodes $u_1, \ldots, u_{m-1}, u_{m+1}, \ldots, u_p$, instead of the nodes $u_1, \ldots, u_{m-1}, u_{m+1}, \ldots, u_p$, to be children of the node u_m. This does not affect the correctness of the algorithm since *level*$(\pi_i; u_m) = h$ and *level*$(\pi_i; u_j) = h + 1$ for every j, where $j \neq m$ and $1 \leqslant j \leqslant p$. Thus, the computation of finding the node u_m can be done in $O(\log n)$ time with $O(nb_i/\log n + b_i)$ processors, where $b_i < n$.

Thus, the operations performed in Step 7.1 during an iteration of the algorithm are executed in $O(\log n)$ time using a total of $O(n(b_i/\log n) + nb_i/\log n + b_i)$ processors, where $0 \leqslant i \leqslant n$ and $b_i < n$. Since $(b_1 + b_2 + \cdots + b_k) < n$, where $k < n$ (see single and multiple link cases), we conclude that Step 7.1 is executed in $O(\log n)$ using a total of $O(n^2/\log n)$ processors on the CREW PRAM model.

Based on the analysis of the time and processor complexity of Step 7.1, it is easy to see that we can paint black the appropriate nodes of all the active trees within the same time and processor complexity, and, thus, Step 7.2 is executed in $O(\log n)$ using a total of $O(n^2/\log n)$ processors on the CREW PRAM model. (We should point out that, in Step 7.1 the matrix B_i is of size $(b_i + 1) \times b_j$, where b_j is the number of nodes in the set *active-link-nodes*$(T[\pi_i])$, while in Step 7.2 the matrix B_i is of size $(b_i+1)\times n$.)

Let us now show how we can implement the removal operation performed in Step 7.3. Let $T[\pi_i]$ be an active tree and let u be a white node such that its parent $p(u)$ is a black node. Then, make the parent of the node u to be the node w, where w is the white node such that $w=p(u)$. This operation can be done in $O(\log n)$ with $O(n^2/\log n)$ processors on the CREW PRAM, for all the active trees $T[\pi_i]$, $0\leqslant i\leqslant n$. Consequently, find the black node u of the active tree $T[\pi_i]$, such that its parent $p(u)$ is a white node and remove the subtree of $T[\pi_i]$ rooted at u, $0\leqslant i\leqslant n$; this subtree contains only black nodes. Thus, we can easily conclude that Step 7.2 is executed in $O(\log n)$ using a total of $O(n^2/\log n)$ processors on the CREW PRAM model.

From the step-by-step analysis, we have that the whole step is executed in $O(\log n)$ time using a total of $O(n^2/\log n)$ processors of the CREW PRAM model. Thus, the lemma holds. □

We are now in a position to compute the time and processor complexity of algorithm cp-Tree. The complexity of each step of the algorithm is analyzed as follows:

The algorithm consists of eight steps.

Step 1: By Theorem 6.2, the construction of the ds-trees of a permutation π over the set N_n can be done in $O(\log n)$ time using $O(n^2/\log n)$ processors on the CREW PRAM model.

Step 2: This step incorporates operations whose time and processor complexity are given by Lemma 6.1. Thus, it is executed in $O(\log n)$ time with $O(n^2/\log n)$ processors on the EREW PRAM model.

Step 3: Obviously, this step can be executed in $O(1)$ sequential time.

Step 4: The union of $n+1$ sets, each of length $O(n)$, is performed in $O(\log n)$ time with $O(n^2/\log n)$ processors on the EREW PRAM model.

Step 5: Obviously, this step is executed in $O(\log n)$ with $O(n/\log n)$ processors on the CREW PRAM model.

Step 6: Lemma 6.2 provides us with the time and processor complexity of Step 6 of the algorithm. Thus, we have that this step is executed in $O(\log n)$ time with $O(n^2/\log n)$ processors on the CREW PRAM model.

Step 7: From Lemma 6.2 we have that Step 7 is executed in $O(\log n)$ time using a total of $O(n^2/\log n)$ processors on the CREW PRAM model.

Step 8: This step makes the link-nodes, active-link-nodes and copy-active-nodes sets of all the active trees to be empty sets. Obviously, this initialization step can be executed in $O(\log n)$ time with $O(n^2/\log n)$ processors on the EREW PRAM model.

From Theorem 5.1, we have that algorithm cp-Tree performs $O(\log n)$ iteration. Thus, taking into consideration the time and processor complexity of each step of the algorithm, we present the following result.

Theorem 6.3. *Given a permutation π over the set 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

We can easily see that algorithm `Coloring` incorporates all the operations described in the previous algorithms. More precisely, by Theorem 6.3 we have that Step 1 takes $O(\log^2 n)$ time and $O(n^2/\log n)$ processors on the CREW PRAM model. The level of each n-node tree is computed in $O(\log n)$ time with $O(n/\log n)$ processors on the EREW PRAM model. Finally, 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 a permutation graph 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 study the problem of coloring permutation graphs using certain properties of the lattice representation of a permutation and relationships between permutations, directed acyclic graphs and rooted trees. Given a permutation π over the set N_n, we propose an efficient parallel algorithm which colors the n-node permutation graph $G[\pi]$ in $O(\log^2 n)$ time using a total of $O(n^2/\log n)$ processors on the CREW PRAM model.

Our algorithm is motivated by the work of Yu and Chen [22]. They presented an algorithm which solves the coloring problem on a permutation graph $G[\pi]$ by transforming the permutation π it into a set of planar points, constructing an acyclic directed graph, and solving the largest-weight path problem on this acyclic digraph. Their algorithm runs in $O(\log^2 n)$ time with $O(n^3/\log n)$ processors on the CREW PRAM model of computation, or in $O(\log n)$ time with $O(n^3)$ processors on the CRCW PRAM. In this paper we solve the same coloring problem by using a different approach; our algorithm constructs a rooted tree, known here as cp-tree $T^*[\pi]$, using certain combinatorial properties of π, and then solves the coloring problem on $G[\pi]$ by computing the level function on the tree $T^*[\pi]$.

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; that is, in $O(\log^2 n)$ time with $O(n^2/\log n)$ processors; see [8,15,22].

It is worth noting that the problem of finding out the longest common subsequence of two words is a variant of the problem of finding a maximum independent set on a permutation graph; see [12]. Thus, Lu's paper might be a good indicator how to solve also coloring and other problems on permutation graphs.

In closing, we mention that many aspects of our coloring algorithm apply for comparability graphs [6,17]. One can define as the ds-tree of a node x a tree $T[x]$ with root x and containing all the nodes $u < x$ (with respect to the partial order $<$); the parent of any node y in $T[x]$ is some successor of y in the transitive reduction over the order $<$. The notion of link-nodes and active-link nodes applies also for comparability graphs. Thus, one can formulate all steps of the coloring algorithm for comparability graphs and, then, study the time and processor complexity.

Acknowledgements

The author would like to thank the anonymous referees for comments and suggestions which improved the presentation of this paper.

References

[1] M. Andreou, S.D. Nikolopoulos, NC coloring algorithms for permutation graphs, Nordic J. Comput. 6 (1999) 422–445.
[2] M.J. Atallah, G.K. Manacher, J. Urrutia, Finding a minimum independent dominating set in a permutation graph, Discrete Appl. Math. 21 (1988) 177–183.
[3] P. Beame, J. Hastad, Optimal bounds for decision problems on the CRCW PRAM, J. Assoc. Comput. Mach. 36 (1989) 643–670.
[4] A. Brandstadt, D. Kratsch, On domination problems for permutation and other graphs, Theoret. Comput. Sci. 54 (1987) 181–198.
[5] M. Farber, J.M. Keil, Domination in permutation graphs, J. Algorithms 6 (1985) 309–321.
[6] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, Inc., New York, 1980.
[7] D. Helmbold, E.W. Mayr, Applications of parallel algorithms to families of perfect graphs, Computing 7 (1990) 93–107.
[8] J. JáJá, An Introduction to Parallel Algorithms, Addison-Wesley, Inc., Reading, MA, 1992.
[9] T.R. Jensen, B. Toft, Graph Coloring Problems, Wiley, New York, 1995.
[10] D. Kozen, U.V. Vazirani, V.V. Vazirani, NC Algorithms for Comparability Graphs, Interval Graphs, and Testing for Unique Perfect Matching, Lecture Notes in Computer Science, Vol. 206, Springer, Berlin, 1985, pp. 498–503.
[11] J.Y.-T. Leung, Fast algorithms for generating all maximal independent sets of interval, circular-arc and chordal graphs, J. Algorithms 5 (1984) 22–35.
[12] M. Lu, Parallel computation of longest-common-subsequence, Advances in Computing and Information, ICCI'90, Proceedings of International Conference, Niagara Falls, Canada, 1990, Lecture Notes in Computer Science, Vol. 468, 1991, pp. 385–394.
[13] S.D. Nikolopoulos, Ch. Papadopoulos, On the performance of the first-fit coloring algorithm on permutation graphs, Inform. Process. Lett. 75 (2000) 265–273.
[14] A. Pnueli, A. Lempel, S. Even, Transitive orientation of graphs and identification of permutation graphs, Canadian J. Math. 23 (1971) 160–175.
[15] J. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.
[16] R. Sedgewick, P. Flajolet, An Introduction to the Analysis of Algorithms, Addison-Wesley, Inc., Reading, MA, 1996.
[17] J. Spinrad, On comparability and permutation graphs, SIAM J. Comput. 14 (1985) 658–670.
[18] J. Spinrad, A. Brandstadt, L. Stewart, Bipartite permutation graphs, Discrete Appl. Math. 18 (1987) 279–292.
[19] K.J. Supowit, Decomposing a set of points into chains, with applications to permutation and circle graphs, Inform. Process. Lett. 21 (1985) 249–252.

[20] K.H. Tsai, W.L. Hsu, Fast algorithms for the dominating set problem on permutation graphs, International Symposium SIGAL ’90, Tokyo, Japan, 1990, Lecture Notes in Computer Science, Vol. 450, Springer, Berlin, 1990, pp. 109–117.
[21] S. Tsukiyama, M. Ide, H. Ariyoshi, I. Shirakawa, A new algorithm for generating all the maximal independent sets, SIAM J. Comput. 6 (1977) 505–517.
[22] C.-W. Yu, G-H. Chen, Parallel algorithms for permutation graphs, BIT 33 (1993) 413–419.
[23] C.-W. Yu, G-H. Chen, Generate all maximal independent sets in permutation graphs, Internat. J. Comput. Math. 47 (1993) 1–8.