Parallel Algorithm for Optimal Vertex-Rankings of Permutation Graphs

by

Muhammad Abdul Hakim Newton

October, 2002

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka-1000, Bangladesh

Parallel Algorithm for Optimal Vertex-Rankings of Permutation Graphs

by

Muhammad Abdul Hakim Newton

October, 2002

Submitted to

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka-1000, Bangladesh

in partial fulfillment of the requirement for the degree of

M. Sc. Engineering (Computer Science and Engineering)

PARALLEL ALGORITHM FOR OPTIMAL VERTEX-RANKINGS OF PERMUTATION GRAPHS

A thesis submitted by

MUHAMMAD ABDUL HAKIM NEWTON
Student No. 040005022P
for the partial fulfillment of the degree of
M. Sc. Engineering (Computer Science and Engineering).
Examination held on October 8, 2002.

Approved as to style and contents by:

[illegible]

Dr. Md. Abul Kashem Mia Associate Professor & Head Department of CSE, BUET, Dhaka-1000, Bangladesh.	Chairman, Supervisor, and Ex-officio

[illegible]

Dr. M. Kaykobad Professor Department of CSE, BUET, Dhaka-1000, Bangladesh.	Member

[illegible]

Dr. Chowdhury Mofizur Rahman Professor Department of CSE, BUET, Dhaka-1000, Bangladesh.	Member

[illegible]

Dr. Md. Kamrul Hasan Associate Professor Department of EEE, BUET, Dhaka-1000, Bangladesh.	Member (External)

Certificate

This is to certify that this thesis work has been done by ***Muhammad Abdul Hakim Newton***, *Student No. 040005022p*, under the supervision of ***Dr. Md. Abul Kashem Mia***, *Associate Professor and Head, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka-1000, Bangladesh.* It is also declared that neither this thesis nor any part of it has been submitted or is being submitted to anywhere else for the award of any degree or diploma.

[illegible]
Muhammad Abdul Hakim Newton

[illegible]
Dr. Md. Abul Kashem
Supervisor of the Thesis

Contents

List of Figures

Abstract

In this thesis, we give a parallel algorithm to find the c-vertex-ranking of a permutation graph. A c-vertex-ranking of a graph $G = \langle V, E \rangle$, for a positive integer c, is a labeling of the vertices of G such that, for any label k, deletion of all vertices with labels greater than k leaves connected components, each having at most c vertices with label k. A c-vertex-ranking is optimal if the number of labels used is as small as possible. The c-vertex-ranking problem is to find an optimal c-vertex-ranking of a given graph. A graph $G = \langle V, E \rangle$ is a permutation graph if $V = \{1, 2, \ldots, n\}$ and $E = \{\langle i, j \rangle \mid (i - j)*(\pi^{-1}(i) - \pi^{-1}(j)) < 0\}$, where $\pi = [\pi(1), \pi(2), \ldots, \pi(n)]$ is a permutation of the numbers $1, 2, \ldots, n$ and $\pi^{-1}(i)$ is the position of i in the permutation. Deogun *et al.* gave a sequential algorithm to solve the optimal 1-vertex-ranking problem on permutation graphs using time $O(n^6)$, where n is the number of vertices in the graph. Later, we have solved the same problem in $O(n^3)$ sequential time. But, there is no known parallel algorithm to solve the c-vertex-ranking problem for permutation graphs. In this thesis, we have devised a parallel algorithm for the same problem with any value of c that runs in $O(\log^2 n)$ parallel time using $O(n^3 \log n)$ operations on the CREW PRAM model.

Chapter 1 Introduction

This chapter introduces the vertex-ranking problem and their applications. It also highlights on the complexities of the vertex-ranking problem on several types of graphs.

1.1 Graph

A *graph* $G = \langle V, E \rangle$ is an ordered pair consisting of a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a set of edges $E = \{e_1, e_2, \ldots, e_m\}$. Here, $|V| = n$ and $|E| = m$. An *edge* $e = \langle u, v \rangle$ with $u, v \in V$ is an unordered pair and is said to be *incident* on the vertices u and v. Figure 1.1 depicts a graph with four vertices and five edges. The circles represent the vertices and the lines represent the edges.

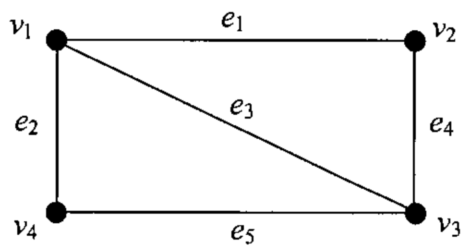


Figure 1.1 A graph with 4 vertices and 5 edges.

1.2 Vertex-Ranking

A *c-vertex-ranking* of a graph $G = \langle V, E \rangle$, for any positive integer c, is a labeling of the vertices of G with integers such that, for any label k, deletion of all vertices with label greater than k leaves connected components, each having at most c vertices with label k [ZNN95]. The integer label of a vertex is called its *rank*. The *ordinary vertex-ranking* of a graph G is a c-vertex-ranking if $c = 1$, and is defined alternatively as a labeling of the vertices of G with positive integers such that, every path between any two vertices with same label contains a vertex with a larger label [IRV88]. The 1-vertex-ranking shown in

Figure 1.2 a) is invalid because deletion of the vertices with label greater than 3 leaves one connected component having 2 vertices with label 3. On the other hand, The 1-vertex-ranking shown in Figure 1.2 b) is valid because deletion of all vertices with label greater than any k leaves connected components having at most one vertex with rank k.

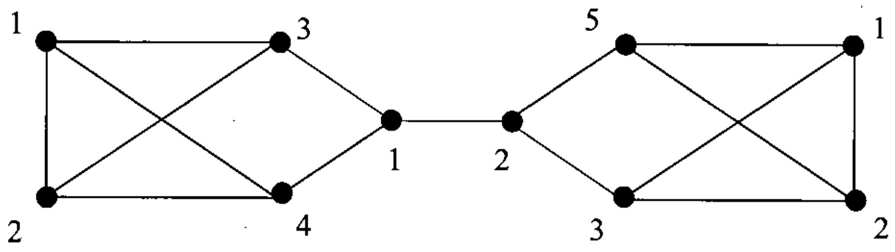


Figure 1.2 a) An invalid 1-vertex-ranking

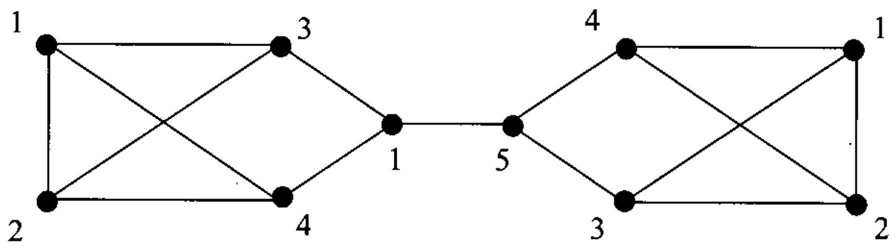


Figure 1.2 b) A valid 1-vertex-ranking

1.3 Vertex-Ranking Problem

Given a graph G and an integer c, a c-vertex-ranking of G that uses the minimum number of ranks is called an *optimal c-vertex-ranking* of G. The number of ranks used in an optimal c-vertex-ranking of G is called the *c-vertex-ranking number* of G and is denoted by $r_c(G)$. The *c-vertex-ranking problem* is to find an optimal c-vertex-ranking of G [ZNN95]. Figure 1.3 a) depicts a non-optimal 1-vertex-ranking using 6 ranks and Figure 1.3 b) shows an optimal 1-vertex-ranking of a graph using 5 ranks which is minimum. There is no valid 1-vertex-ranking for the graph in Figure 1.3 b) that uses less than 5 ranks.

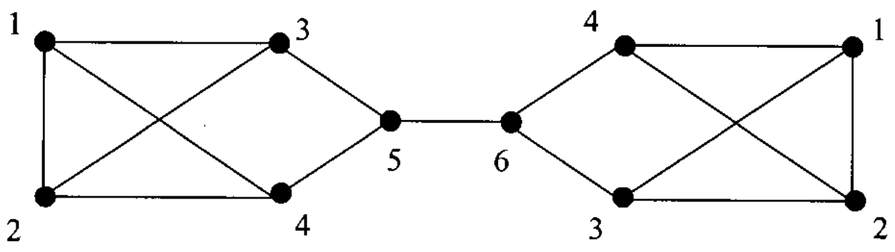


Figure 1.3 a) A non-optimal 1-vertex-ranking

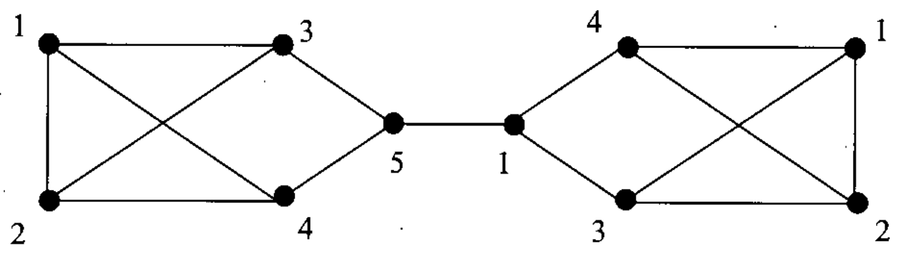


Figure 1.3 b) An optimal 1-vertex-ranking

1.4 Applications of Vertex-Ranking Problem

The vertex-ranking problem has received much attention because of the growing number of applications. For example, it plays important roles in VLSI (Very Large Scale Integration) layout design [L80, SDG92] and in scheduling the parallel assembly of a complex multi-part product from its component [IRV88]. The vertex-ranking problem has also an important application in Cholesky factorization of matrices. Let $Ax = b$ be a large system of linear equations, where A is an n-by-n sparse symmetric matrix. Solving such a system via Cholesky factorization involves computing a lower triangular matrix L such that $A = LL^T$, and then solving the systems $L^T x = y$ and $Ly = b$. Let A_1 be a matrix obtained from A by replacing each non-zero element with 1. Let G be a graph with adjacency matrix A_1. One may split A_1 into smaller matrices (which can be handled in parallel) by removing some elements on the main diagonal of A_1 together with their rows and columns. Clearly, these rows and columns correspond to a separator S of G, where a separator is defined as a set of vertices whose removal from the graph separates some two vertices in different connected components. The same procedure is applied recursively to the connected components of $G - S$. Thus, the optimal vertex-ranking of G corresponds to a Cholesky factorization of A having the minimum recursive depth [DR83, L90].

1.5 Known Results on Vertex-Ranking Problem

The vertex-ranking problem is NP-hard in general [P88], that is, it is very unlikely to have a polynomial-time algorithm to solve the vertex-ranking problem for general graphs. Hence, graphs are classified using some characteristic properties and sometimes these properties are found to be very helpful in vertex-ranking of that class of graphs. Using this strategy, vertex-ranking problem has been solved in polynomial-time for several classes of graph. For example, Iyer *et al.* presented an $O(n\log n)$ time sequential algorithm to solve the vertex-ranking problem for trees [IRV88], where n is the number of vertices of the given tree. Then Schäffer obtained a linear time sequential algorithm by refining their algorithm and its analysis [S89]. Later, Rahman *et al.* presented an $O(\log n)$ time parallel algorithm using $O(n/\log n)$ processors on the EREW PRAM model [RK01]. There is a polynomial-time sequential algorithm and an $O(\log n)$ time polynomial processor parallel algorithm presented by Kashem *et al.* which solves the vertex-ranking problem for partial k-trees, that is, graphs of treewidth bounded by a fixed integer k [KZN00]. Recently Deogun *et al.* gave sequential algorithms to solve the vertex-ranking problem for interval graphs in $O(n^3)$ time and for permutation graphs in $O(n^6)$ time [DKKM94]. Later, we have solved the ordinary vertex-ranking problem for permutation graphs in $O(n^3)$ sequential time [NK99].

There is no known parallel algorithm to solve the c-vertex-ranking problem for permutation graphs. In this thesis, we devised a parallel algorithm for the same problem based on our sequential algorithm and it runs in $O(\log^2 n)$ time using $O(n^3\log n)$ operations on the CREW PRAM (Concurrent Read Exclusive Write Parallel Random Access Machine) model.

Graph classes	**Algorithm**	**Complexity**	**Reference**
Trees	Sequential	$O(n)$ time	[S89]
	Parallel	$O(\log n)$ time using $O(n/\log n)$ processor on the EREW PRAM	[RK01]
Partial k-trees	Sequential	Polynomial time	[KZN00]
	Parallel	$O(\log n)$ time polynomial processor on the PRAM	[KZN00]
Interval graphs	Sequential	$O(n^3)$ time	[DKKM94]
Permutation graphs	Sequential	$O(n^6)$ time	[DKKM94]
	Sequential	$O(n^3)$ time	[NK99]
	Parallel	$O(\log^2 n)$ time using $O(n^3\log n)$ operations on the CREW PRAM	Ours

Table 1.1 Known results on vertex-ranking problem

Chapter 2 Preliminaries

This chapter defines some basic terminology of graphs, subgraphs, connected components and separators, permutations and permutation graphs, parallel random access machine etc.

2.1 Graphs

A *graph* $G = \langle V, E\rangle$ is an ordered pair consisting of a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a set of edges $E = \{e_1, e_2, \ldots, e_m\}$. Here, $|V| = n$ and $|E| = m$. An *edge* $e = \langle u, v\rangle$ with u, $v \in V$ is an unordered pair and is said to be *incident* on the vertices u and v. The vertices u and v are said to be *adjacent* to each other. We also say u and v are *neighbor* of each other. The *degree* of a vertex v in a graph G is the number of edges incident to the vertex v. A vertex with degree 0 is called an *isolated vertex*. A *path* $P = \langle w_1, w_2, \ldots, w_k\rangle$ from w_1 to w_k in G is a sequence of vertices such that every edge $\langle w_i, w_{i+1}\rangle \in E$ where $1 \le i < k$. Figure 2.1 depicts a graph with four vertices and five edges. The circles represent the vertices and the lines represent the edges. Here, $e_1 = \langle v_1, v_2\rangle$ is an edge joining the vertices v_1 and v_2 and hence v_1 and v_2 are adjacent and also, neighbor to each other. Moreover, the vertex v_1 has degree 3.

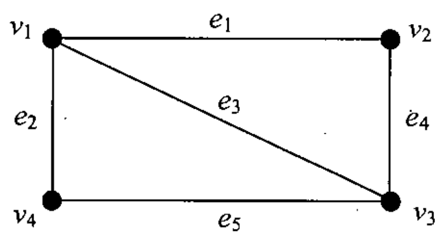


Figure 2.1 A graph with 4 vertices and 5 edges.

2.2 Subgraphs

A graph $G_1 = \langle V_1, E_1 \rangle$ is a *subgraph* of a graph $G = \langle V, E \rangle$ if $V_1 \subseteq V$ and $E_1 \subseteq E$; we denote this by $G_1 \subseteq G$. If $E_1 = E \cap (V_1 \times V_1)$ then G_1 is said to be a *subgraph of G induced by* V_1 and is denoted by $G[V_1]$. Here, $G[V_1]$ is obtained by deleting the vertices not in V_1 along with all the edges incident on them. If we eliminate vertex v_3 along with every edge incident on it from the graph shown in Figure 2.1, we get a subgraph induced by the set $\{v_1, v_2, v_4\}$ which is shown in Figure 2.2.

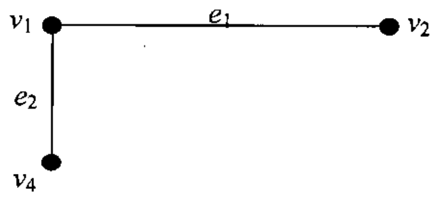
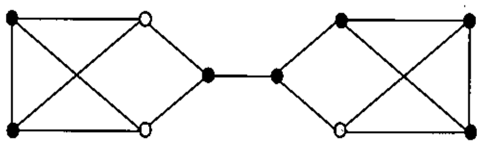


Figure 2.2 A subgraph of the graph in Figure 2.1 induced by $\{v_1, v_2, v_3\}$.

2.3 Connected Graphs and Separators

A graph G is *connected* if there is a path between any pair of distinct vertices of the graph; otherwise it is *disconnected.* A *connected component* of a graph is a maximal connected subgraph. A *separator* of a connected graph is a set of vertices whose deletion makes G disconnected.



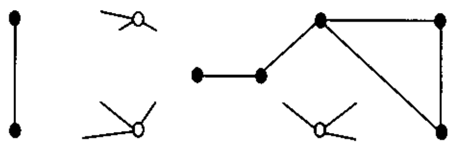


Figure 2.3 a) A connected graph **Figure 2.3** b) A disconnected graph

Figure 2.3 a) and b) shows a connected graph and a disconnected graph respectively. The hollow vertices in Figure 2.3 a) are in a separator because deleting the vertices we get a disconnected graph but the shaded vertex is not in a separator since its deletion does not disconnect the graph.

2.4 Permutation Graphs

2.4.1 Permutation

A *permutation* is a rearrangement of the positions of a sequence of numbers. We think a permutation of the numbers 1, 2, . . ., n as the sequence $\pi = [\pi(1), \pi(2), \ldots, \pi(n)]$. Moreover, $\pi^{-1} = [\pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(n)]$ is also a sequence called *Inverse Permutation* where $\pi^{-1}(k)$ denotes the position of k in π.

$$\begin{array}{rcccccccc}
k = & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\pi = & 2 & 6 & 4 & 3 & 8 & 1 & 5 & 7 \\
\pi^{-1} = & 6 & 1 & 4 & 3 & 7 & 2 & 8 & 5
\end{array}$$

Figure 2.4 A sequence of numbers, one of its permutation and the inverse permutation

2.4.2 Permutation Diagram

Let π be a permutation of 1, 2, . . . , n. Write the numbers 1, 2, . . . , n horizontally from left to right. This is the *top row*. Underneath the top row, write the numbers $\pi(1)$, $\pi(2)$, . . . , $\pi(n)$, also, horizontally from left to right. And, this is the *bottom row*. Draw straight-line segments, joining the two 1s, the two 2s, etc. The diagram obtained is called the *permutation diagram* of the given permutation.

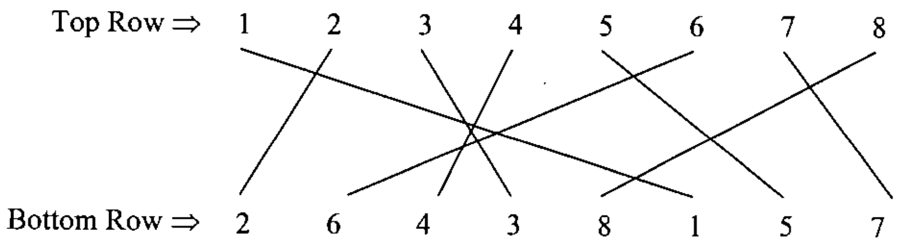


Figure 2.5 The permutation diagram of the permutation shown in Figure 2.4

2.4.3 Permutation Graph

Let π be a permutation of the numbers 1, 2, . . ., n. We can construct a graph $G_\pi = \langle V, E\rangle$, where $V = \{1, 2, \ldots, n\}$ and $E = \{\langle i, j\rangle \mid (i-j)*(\pi^{-1}(i) - \pi^{-1}(j)) < 0\}$. A graph G is a *permutation graph* if there is a permutation π such that, $G \cong G_\pi$. Figure 2.5 shows the permutation graph of the permutation shown in Figure 2.4. Notice that, any two vertices i, j of G_π are adjacent if and only if the line joining the two i intersects the line joining two j

in the permutation diagram. Hence, the permutation diagram can illustrate a permutation graph. The mapping from permutation diagrams to permutation graphs is a bijection.

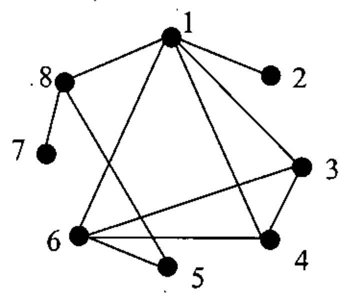


Figure 2.6 The permutation graph of the permutation shown in Figure 2.4

2.5 Parallel Random Access Machine (PRAM)

There are several parallel computation models. Considering the relevance with this thesis, we discuss only the Synchronous Shared Memory Model, also called the Parallel Random Access Machine (PRAM) model.

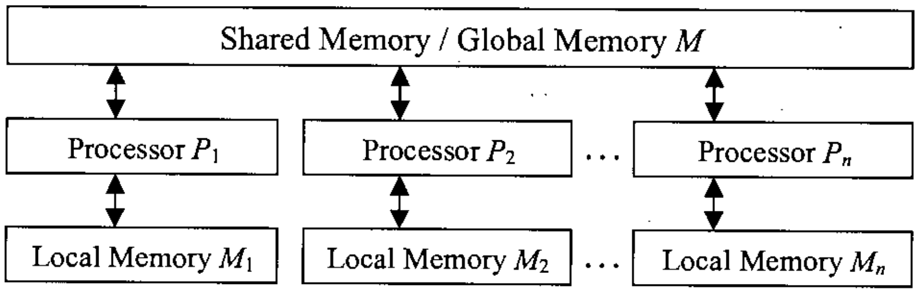


Figure 2.7 Parallel Random Access Machine (PRAM)

Parallel Random Access Machine consists of a number of processors, each of which has its own local memory and can execute its own local program with its own data. All the processors operate synchronously under the control of a common clock and all of them communicate with each other by exchanging data through a shared memory unit referred to as the *global memory*. Each processor is uniquely identified by an index, called the *processor number* or *processor id*, which is available locally and hence, can be referred to in the processor's program.

There are several variations of the PRAM model based on simultaneous access of several processors to the same location of the global memory. The *Exclusive Read Exclusive Write* (EREW) PRAM does not allow any simultaneous access to a single memory location. The *Concurrent Read Exclusive Write* (CREW) PRAM allows simultaneous access for read instructions only. Access to a location for both read and write instructions is allowed in the *Concurrent Read Concurrent Write* (CRCW) PRAM. The three principal varieties of CRCW PRAMs are differentiated by how concurrent writes are handled. The *common* CRCW PRAM allows concurrent writes only when all processors are attempting to write the same value. The *arbitrary* CRCW PRAM allows an arbitrary processor to succeed. The *priority* CRCW PRAM assumes that the indices of the processors are linearly ordered and allows the one with minimum index to succeed.

Chapter 3 Vertex-Ranking of General Graphs

This chapter dicusses on vertex-ranking of general graphs. Section 3.1 presents some definitions and easy lemmas on vertex-ranking of general graphs. The vertex-separator tree and its association with the vertex-ranking problems are illustrated in Section 3.2. The minimal separator concept and a generalized vertex-ranking procedure based on minimal separators along with its proof of correctness are devised in Section 3.3.

3.1 Vertex-Ranking of Graphs

Though in Chapter 1, we have discussed vertex-ranking and vertex-ranking problem but for the sake of completeness of this chapter they are more formally defined as follows.

Definition 3.1.1 Given a graph $G = \langle V, E \rangle$, a *c-vertex-ranking* of G, for any positive integer c, is a labeling φ of the vertices of G with integers such that, for any label k, deletion of all vertices with labels greater than k leaves connected components, each having at most c vertices with label k [ZNN95]. The integer label of a vertex v is called its *rank* and is denoted by $\varphi(v)$. The *ordinary vertex-ranking* of a graph G is a c-vertex-ranking if $c = 1$, and is defined alternatively as a labeling of the vertices of G with positive integers, such that, every path between any two vertices with the same label contains a vertex with a larger label [IRV88]. A c-vertex-ranking of G that uses the minimum number of ranks is called an *optimal c-vertex-ranking* of G. The number of ranks used in an optimal c-vertex-ranking of G is called the *c-vertex-ranking number* of G and is denoted by $r_c(G)$. The *c-vertex-ranking problem* is to find an optimal c-vertex-ranking of G.

Remark 3.1.2 For a complete graph K_n with n vertices, the *c-vertex-ranking number* $r_c(K_n) = \lceil n/c \rceil$ and the c-vertex-ranking number of a disconnected graph is equal to the maximum c-vertex-ranking number of its components.

3.2 Vertex-Elimination Tree

The concept of vertex-elimination tree is very important for the vertex-ranking problem.

Definition 3.2.1 Let $G = \langle V, E\rangle$ be a connected graph. A *vertex-elimination* tree for G is a rooted tree T with vertex set V defined recursively as follows. If $V = \{v\}$ then T is the rooted tree containing only one vertex v. Otherwise, choose a vertex $r_T \in V$ as the root of T. Let $C_1, C_2, \ldots, C_q$ be the connected components of $G[V \setminus \{r_T\}]$. For each component C_i, let T_i be a vertex-elimination tree. T is defined by making each root r_{T_i} of T_i adjacent to r_T. The height of the rooted tree T is the maximal length of a path from the root to a leaf.

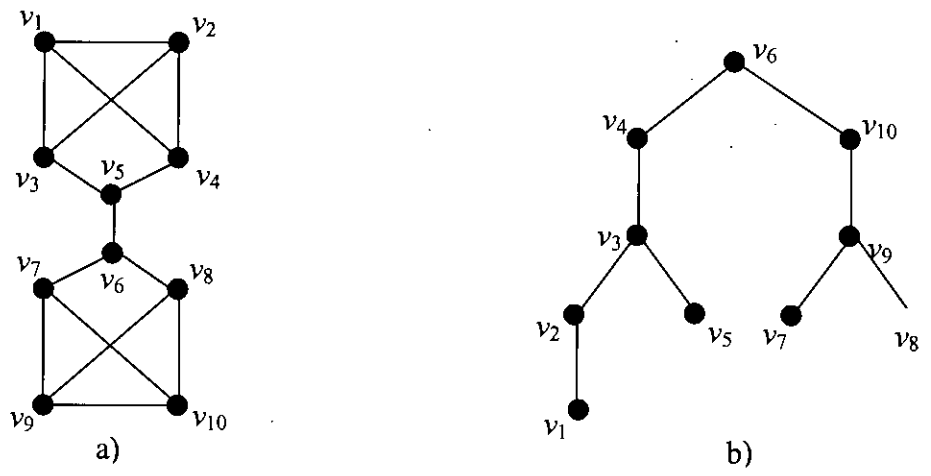


Figure 3.1 a) A graph, and b) one of its vertex-elimination tree

Definition 3.2.2 A path $P = \langle w_1, w_2, \ldots, w_k\rangle$ in a given tree T is a *linear descending path* from w_1 to w_k if each w_l $(1 \leq l < k)$ has only one child w_{l+1}. A path containg a single vertex is vacuously linear descending. P is a *maximal linear descending path* if no linear descending path $Q \neq P$ contains P.

The following procedure describes a method to assign ranks to the vertices of a graph using a vertex-elimination tree of the graph.

Procedure 3.2.3 Let T be a vertex-elimination tree of a given graph G and T_i $(1 \leq i \leq q)$ be one of the q subtrees found by deleting the vertex v from the subtree of T rooted at v. If $c_{k,i}$ denotes the frequency of rank k in the subtree T_i and k_i denotes the maximum rank used in the tree T_i and $\max_{1\le i\le q} k_i$ is denoted by k then a c-vertex-ranking φ can be obtained by the following bottom-up scheme–

$$\varphi(v) = \begin{cases} 1, \text{ if } v \text{ is a leaf vertex of } T \text{ or } n = 1 \\ k, \text{ if } v \text{ is an internal vertex of } T \text{ and } \sum_{1\le i\le q} c_{k,i} < c \\ k+1, \text{ if } v \text{ is an internal vertex of } T \text{ and } \sum_{1\le i\le q} c_{k,i} \ge c \end{cases}$$

Remark 3.2.4: Since the rank assignment is a monotonically increasing bottom-up function, the root of any subtree gets the maximum rank used in that subtree. Besides frequency of no rank less than the maximum rank is less than c.

It needs to prove that Procedure 3.2.3 assigns valid ranks to the vertices of the graph and Lemma 3.2.5 does this.

Lemma 3.2.5 *For any $c \in N$, Procedure 3.2.3 leads to a valid c-vertex-ranking of a given graph G.*

Proof: The proof is very simple and straightforward. A vertex on a path from any other vertex to the root gets larger rank if it is closer to the root. So, deletion of a set of vertices with ranks greater than k always deletes all the nodes on the path from every v with $\varphi(v) = k + 1$ to the root. Lastly, the assignment process ensures that no subtree gets a rank more than c times. □

We found a way to assign valid ranks. Now, how this can be done optimally. The following lemma shows that applying Procedure 3.2.3 on a minimum-height vertex-elimination tree of G results into an optimal c-vertex-ranking.

Lemma 3.2.6 *Let T be a vertex-elimination tree with minimum height of a connected graph G. Then, for any $c \in N$, applying Procedure 3.2.3 on T leads to the optimal c-vertex-ranking of G and $r_c(G) = \varphi(r)$ where r is the root of T.*

Proof: The proof is by induction on the height of the tree h. For $h = 0$, the assertion is trivially true. By induction hypothesis, assume that the assertion is true for any graph G

having a vertex-elimination tree with minimum height less than h. In induction step, delete the root r_T from T resulting into q subtrees $T_1, T_2, \ldots, T_q$. Also delete r_T from G resulting into q connected components $G_1, G_2, \ldots, G_q$. By definition T_i $(1 \leq i \leq q)$ is the vertex-elimination tree of G_i with height less than h. By optimality principle, any subtree of a tree with minimum height is also of minimum height. So, each G_i can be ranked optimally by Procedure 3.2.3. Let k_i be the maximum rank used in T_i and the frequency of k in T_i is $c_{k, i}$. Clearly, $r_c(G_i) = k_i = \varphi(r_{T_i})$, where r_{T_i} is the root of T_i. Vertex r_T can not get a rank $k < k_m = \max_{1 \leq i \leq q} k_i$. This is because r_T is adjacent to some vertices of each T_i and so to those vertices in G_i, and deletion of the vertices with rank greater than k in G would leave more than c vertices with rank k (see Remark 3.2.4). Therefore, assign a rank k_m to r_T if $\sum_{1 \leq i \leq q} c_{k_m, i} < c$; otherwise assign $k_m + 1$ because the vertices in T are in a single component of G. Hence, r_T is ranked with the minimum rank possible. □

3.3 Vertex-Separator and Vertex-Separator Tree

Let us first see the definition of vertex-separator, which is also discussed in chapter 2. Then, we find a close relation between the vertex-elimination tree and the vertex-separator tree.

Definition 3.3.1 A subset $S \subseteq V$ is a (u, v) *separator* (or in general *vertex-separator*) for two nonadjacent vertices u and v of a connected graph G, if the removal of S separates u and v into two distinct connected components. If S is a (u, v) separator and no proper subset of S is a (u, v) separator then S is a *minimal* (u, v) *separator* (or in general *minimal vertex-separator*). A minimal vertex-separator that is not properly contained in any other minimal vertex-separator is called an *inclusion minimal vertex-separator.*

Definition 3.3.2 Let G be a connected graph. A *vertex-separator tree* for G is a rooted tree T with the vertex set V defined recursively as follows. Choose a separator S as the root of T. If $S \neq V$ then assume $G_1, G_2, \ldots, G_q$ be the connected components of $G[V \setminus S]$. For each such connected component G_i $(1 \leq i \leq q)$, let T_i be a vertex-separator tree. Construct T by making each root S_i of T_i adjacent to S.

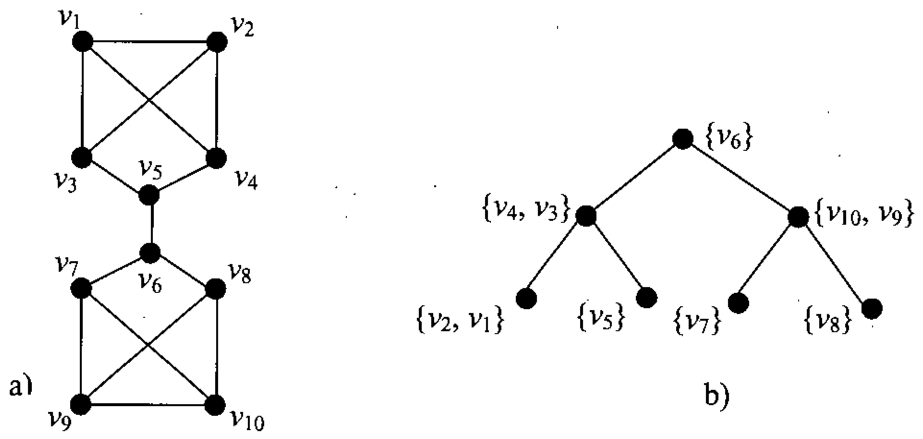


Figure 3.2 a) A graph, and b) one of its vertex-separator tree

The following observation shows the correspondence between vertex-elimination tree and vertex-separator-tree.

Observation 3.3.3 Vertices of any maximal linear descending path in a vertex-elimination tree of a graph G is a vertex-separator of G. So by merging the vertices of the maximal linear descending path into a single node converts a vertex-elimination tree into a vertex-separator tree. On the contrary, expanding a vertex-separator into a linear descending path converts a vertex-separator tree into a vertex-elimination tree.

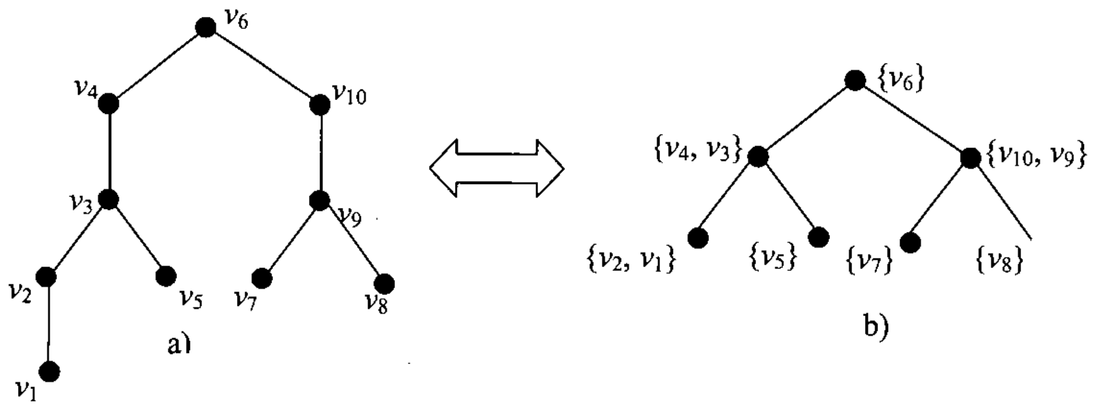


Figure 3.3 a) A vertex-elimination tree, and b) its corresponding vertex-separator tree

The following observation is very important for ranking the vertices of a graph using a vertex-separator tree.

Observation 3.3.4 In order to obtain an optimal c-vertex-ranking of a graph G by Procedure 3.2.3, we need a vertex-elimination tree with minimum height. The height of a vertex-elimination tree can be reduced if the lengths of the linear descending paths are shorten. This is possible if we use minimal vertex-separators to construct a vertex-separator tree. Hence, we conclude that a vertex-elimination tree with minimum height can be obtained from the vertex-separator trees constructed dynamically and only with minimal separators.

Procedure 3.3.5 A straightforward conversion of Procedure 3.2.3 leads to a vertex-ranking scheme based on vertex-separator tree.

i) If S is a leaf node in the vertex-separator tree, then rank the first c vertices of S with 1, then, next c vertices with 2 and so on.

ii) If S is a nonleaf node then let k be the maximum rank used in the descendent subtrees of S and c_k is the sum of the frequencies of rank k in those subtrees. Rank $c - c_k$ vertices with k if $c > c_k$, then rank c vertices with $k + 1$, next c vertices with $k + 2$ and so on.

The following theorem is the main tool of our algorithm. It is a variation of a similar theorem found in [DKKM94].

Theorem 3.3.6 *Let $G = \langle V, E\rangle$ be a graph both connected and not complete. Assume that $k = \max_{1\le i\le q} r_c(G_i)$, $c_k = \sum_{1\le i\le q} c_{k,i}$ and $k' = \lceil |S|/c \rceil$ if $c_k > c$, $k' = \lceil (|S| - c + c_k)/c \rceil$ if $|S| + c_k > c$, otherwise $k' = 0$, where S is a minimal separator in G, each G_i $(1 \le i \le q)$ is a subgraph of $G[V \setminus S]$ and $c_{k,i}$ is frequency of rank k in G_i. Then, $r_c(G) = \min_S (k + k')$.*

Proof: This theorem directly follows from Procedure 3.2.3, 3.3.5, Lemma 3.2.6 and Observation 3.3.3. □

Chapter 4 Permutation Graphs

This chapter thoroughly discusses permutation graphs. Section 4.1 defines the permutation graphs and the permutation diagrams. Section 4.2 illustrates the properties of the line segments and the scanlines and introduces the candidate scanlines. Section 4.3 discusses the pieces and lastly introduces the candidate pieces.

4.1 Permutation Graphs and Permutation Diagrams

Permutations and permutation graphs are already defined in Chapter 2. Still, for the sake of completeness of this chapter they are discussed here.

Definition 4.1.1 A *permutation* is a rearrangement of the positions of a sequence of numbers. We think a permutation of the numbers 1, 2, . . ., n as the sequence $\pi = [\pi(1), \pi(2), \ldots, \pi(n)]$. Moreover, $\pi^{-1} = [\pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(n)]$ is also a sequence called *Inverse Permutation* where $\pi^{-1}(k)$ denotes the position of k in π.

k =1	2	3	4	5	6	7	8
π =2	6	4	3	8	1	5	7
π^{-1} =6	1	4	3	7	2	8	5

Figure 4.1 A sequence of numbers, one of its permutation and the inverse permutation

Definition 4.1.2 Given a permutation $\pi = [\pi(1), \pi(2), \ldots, \pi(n)]$, we can construct a graph $G_\pi = \langle V, E\rangle$ where $V = \{1, 2, \ldots, n\}$ and the $E = \{\langle i, j\rangle \mid (i - j)*(\pi^{-1}(i) - \pi^{-1}(j)) < 0\}$. An undirected graph G is a *permutation graph* if there is a permutation π such that $G \cong G_\pi$. The graph G_π is sometimes called the *inversion graph* of π.

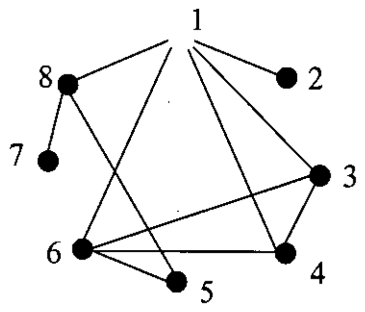


Figure 4.2 The permutation graph of the permutation shown in Figure 4.1

Corollary 4.1.3 Given a permutation π, a permutation graph G_π can be computed in $O(n^2)$ time. Conversely, given a permutation graph G, a permutation π with $G_\pi \cong G$ can also be computed in $O(n^2)$ time.

In this thesis, we assume that the permutation π is given and we identify the permutation graph with the inversion graph G_π.

Definition 4.1.4 Let π be a permutation of 1, 2, . . . , n. Write the numbers 1, 2, . . . , n horizontally from left to right. This is the *top row*. Underneath the top row, write the numbers $\pi(1)$, $\pi(2)$, . . . , $\pi(n)$ also horizontally from left to right. And this is the *bottom row*. Draw straight-line segments, joining the two 1s, the two 2s, etc. The diagram obtained is called the *permutation diagram* of the given permutation.

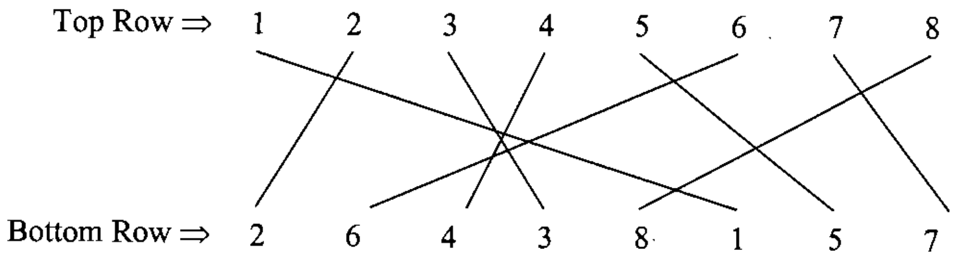


Figure 4.3 Permutation diagram of the permutation shown in Figure 4.1

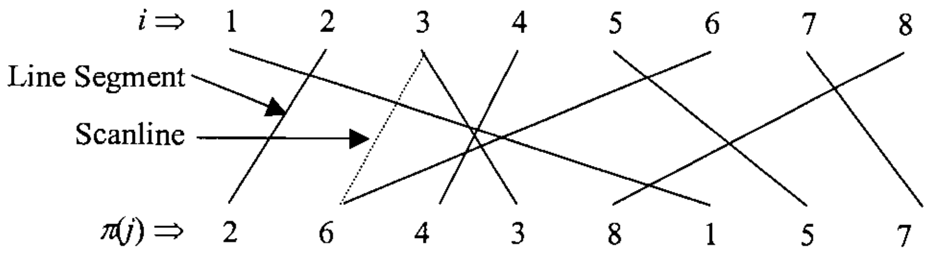
Remark 4.1.5 Any two vertices i, j of G_π are adjacent if and only if the line joining the two i intersects the line joining two j in the permutation diagram. Hence, a permutation graph is an intersection graph, which is illustrated by the permutation diagram. The mapping from permutation diagrams to permutation graphs is a bijection.

From this point, we concentrate on permutation diagrams. It is the permutation diagrams not the permutation graphs, which characterize the features of a permutation. We need to explore the permutation diagram thoroughly to find any kind of useful properties that may be help in the vertex-ranking of permutation graphs.

4.2 Line Segments and Scanlines

Besides the visible straight lines joining some two *k*s, we can imagine some invisible lines in the permutation diagram. These invisible lines join points from the top row with some point in the bottom row.

Definition 4.2.1 A straight-line joining *i* from the top row and $\pi(j)$ from the bottom row is a *line segment* if $i = \pi(j)$ and is denoted by *ith line segment*. If $i \neq \pi(j)$ then it is a *scanline* denoted by *scanline* (i, j). In the permutation diagram, the lines segments are visible and the scanlines are invisible. In Figure 4.4, the solid straight line joining 2 from the top row and $2 = \pi(1)$ from the bottom row is the 2nd line segment and the dashed straight line joining 3 from the top row and $6 = \pi(2)$ from the bottom row is the (3,2) scanline.



$i \Rightarrow$ 1 2 3 4 5 6 7 8
Line Segment
Scanline
$\pi(j) \Rightarrow$ 2 6 4 3 8 1 5 7

Figure 4.4 Line segments and scanlines in the permutation diagram

Corollary 4.2.2 *Number of line segment in a permutation diagram is n and number of scanline is exactly $n^2 - n$, where n is the number of vertices in the permutation graph.*

Observation 4.2.3 Let $s(i, j)$ be a scanline such that $\pi(j) = k$. If the *i*th and *k*th line segment do not intersect each other then the set of line segments intersecting *s* is the (i, k) separator and is denoted by *S*. Notice that, (i, k) and (k, i) separators as we found from the scanlines $(i, \pi^{-1}(k))$ and $(k, \pi^{-1}(i))$ respectively may not be identical. If the *i*th and *k*th line segment intersect each other then, s does not correspond to any separator since i and k are adjacent vertices and hence can not be separated.

Definition 4.2.4 A line segment or a scanline is *to the left (or right) of* another line segment or another scanline if both the top and the bottom of the former is to the left (or right) of those of the later respectively. A line segment or a scanline is *in between* two line segments or two scanlines if both the top and the bottom of the former is in between those of the later two respectively.

Definition 4.2.5 Two scanlines s_1 and s_2 are *equivalent* and denoted by $s_1 \equiv s_2$, if S_1 and S_2 are equal and the set of line segments those are to the left (or right) of the scanline s_1 is equal to the set of line segments those are also to the left (or right) of the scanline s_2.

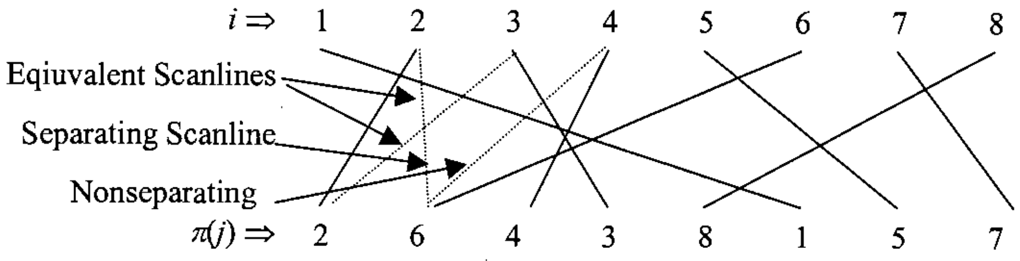

$i \Rightarrow$ 1 2 3 4 5 6 7 8
Eqiuvalent Scanlines
Separating Scanline
Nonseparating
$\pi(j) \Rightarrow$ 2 6 4 3 8 1 5 7

Figure 4.5 Different types of scanlines

Definition 4.2.6 A scanline $s(i,\ \pi^{-1}(j))$ is a *separating scanline* if the ith and jth line segments do not intersect each other; otherwise it is a *nonseparating scanline.* For each separating scanline s, there is a separator S. A separating scanline is an (i, j) *separating scanline* if S is an (i, j) separator.

Observation 4.2.7 If we delete the vertices of S found from a separating scanline $s(i, j)$ with $\pi(j) = k$, the permutation diagram and hence, the permutation graph splits into two separate subgraphs– *left subgraph* and *right subgraph.* The vertices whose corresponding line segments are to the left (or right) of the scanline s are in the left (or right) subgraph. We do not need all the scanlines, we need only the separating scanlines.

Lemma 4.2.8 *Let ith line segment is to the left of jth line segment. All separating scanlines lying in between these line segments are (i, j) separating scanlines. Moreover, all (i, j) separating scanlines are in between the ith and jth line segment.*

Proof. Let $s(x,y)$ be a separating scanlines lying in between the ith and jth line segments and $\pi(y) = z$. Clearly, ith and jth line segments are to the left and right of the scanline s respectively. On the other hand, deletion of the vertices in S separates x and z. From Observation 4.2.7, we see that x is in the left subgraph along with i and z is in the right subgraph along with j. Since the left and the right subgraphs are separate, i and j are also separated.

Say, a scanline s is not in between ith and jth line segment. If s intersects one or both of the two given line segments, then i or j or both are also in separator S and clearly S is not an (i, j) separator. If both ith and jth line segments are to the left (or right) of the scanline s, then both the vertices are in left (or right) subgraph. And, if i and j are in the same connected component, then after deletion all the vertices in S, i and j will be in the same component. Hence S is not an (i, j) separator. □

Our ultimate goal is to identify minimal separators using the scanlines. The following lemma ensures that there is always a scanline for each minimal separator.

Lemma 4.2.9 [K93] *Let G be a permutation graph, and let x and y be nonadjacent vertices in G. For every minimal (x, y) separator S_{xy}, there is a scanline s, which lies in between the line segments corresponding to x and y, such that $S = S_{xy}$.*

Proof: Let S_{xy} be a minimal (x, y) separator and G_x, G_y be the subgraphs of $G[V \setminus S_{xy}]$ containing x and y respectively. Without loss of generality, we may assume that G_x is completely to the left of G_y. Every vertex of S is adjacent to some vertex in G_x and to some vertex of G_y. Notice that, we can choose a scanline s lying in between xth and yth line segment and crossing no line segments of $G[V \setminus S_{xy}]$. We know, all line segments crossing the scanline s must be in S. But, for each element of S, the corresponding line segments must cross s, since it is intersecting a line segment of G_x, which is to the left of s and with a line segment of G_y, which is to the right of s. □

Definition 4.2.10 A separating scanlines s is a *candidate scanline* if S is a minimal separator.

We now devise some procedures to find the candidate scanlines associated with a vertex that is, for a given vertex x, we will find all possible (x, y) or (y, x) separating candidate scanlines, where y is any vertex other than x.

Definition 4.2.11 For any line segment i, we define four types of scanlines though all of them may not exist. Assume that $\pi^{-1}(i) = j$.

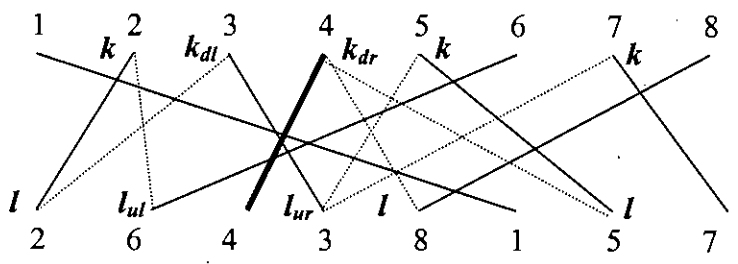


Figure 4.6 Scanlines associated with 4th line segment

Down Left Scanset of the ith line segment, denoted by DLS(i), consists of all scanlines (k_{dl}, l) where k_{dl} is the maximum k satisfying $i \geq k > 1$ and $\pi^{-1}(k-1) < j$; and l satisfies $j - 1 \geq l \geq \pi^{-1}(k_{dl} - 1)$ and $\pi(l) < i$ and $\pi(l+1) \geq i$.

Up Left Scanset of the ith line segment, denoted by ULS(i), consists of all scanlines (k, l_{ul}) where l_{ul} is the maximum l satisfying $j \geq l > 1$ and $\pi(l-1) < i$; and k satisfies $i - 1 \geq k \geq l_{ul} - 1$ and $\pi^{-1}(k) < j$ and $\pi^{-1}(k+1) \geq j$.

Down Right Scanset of the ith line segment, denoted by DRS(i), consists of all scanlines (k_{dr}, l) where k_{dr} is the minimum k satisfying $i \leq k < n$ and $\pi^{-1}(k+1) > j$; and l satisfies $j + 1 \leq l \leq \pi^{-1}(k_{dr} + 1)$ and $\pi(l) > i$ and $\pi(l-1) \leq i$.

Up Right Scanset of the ith line segment, denoted by URS(i), consists of all scanlines (k, l_{ur}) where l_{ur} is the minimum l satisfying $j \leq l < n$ and $\pi(l+1) > i$; and k satisfies $i + 1 \leq k \leq l_{ul} + 1$ and $\pi^{-1}(k) > j$ and $\pi^{-1}(k-1) \leq j$.

The following lemma proves that the four scanlines associated with a vertex as defined in Definition 4.2.11 are indeed candidate scanlines.

Lemma 4.2.12 *All the four types of scanlines associated with a line segment as defined in Definition 4.2.11 are candidate scanlines, i.e., they correspond to some minimal separators.*

Proof: Assume, a scanline (k_{dl}, l) is in the down left scanset of ith line segment. Also, assume $\pi^{-1}(i) = j$. We show that, scanline (k_{dl}, l) is a candidate scanline. Let S_{kl} denotes the separator associated with the scanline (k, l). In Deifinition 4.2.11, l satisfies $j - 1 \geq l$ and $\pi(l) < i$ implying that, $\pi(l)$th line segment is to the left of ith line segment and do not intersect each other; hence there is some separating scanline keeping i and $\pi(l)$ in the right and left subgraph respectively. Again, k_{dl} is the maximum k satisfying $i \geq k \geq 1$ and $\pi^{-1}(k - 1) < j$. Therefore, for all $k \geq k_{dl}$, scanline (k, l) is a (i, l) separating scanline and for all $k > k_{dl}$, $S_{kl} \supset S_{k_{dl}l}$ because $S_{kl} \setminus S_{k_{dl}l} = \{ x: k_{dl} \leq x < k \}$. Scanline (k_{dl}, l) also satisfies $\pi(l + 1) \geq i$ because if a scanline (k_{dl}, l') where $l' > l$ (if there is any) is also in the down left scanset of ith line segment then $S_{k_{dl}l} \setminus S_{k_{dl}l'} \supset \{l'\}$ and $S_{k_{dl}l'} \setminus S_{k_{dl}l} \supset \{(l + 1)\}$; moreover, if $\pi(l + 1) < i$ then $S_{k_{dl}l} \supset S_{k_{dl}(l+1)}$ and $S_{k_{dl}l} \setminus S_{k_{dl}(l+1)} = \{(l + 1)\}$; and lastly, $l \geq \pi^{-1}(k_{dl} - 1) = l'$ (say) because, if $l < l'$ then separator $S_{k_{dl}l} \supset S_{k_{dl}l'}$. Hence, $S_{k_{dl}l}$ is a minimal (i, l) separator and (k_{dl}, l) is a candidate scanline. Similarly, we can show that any scanline which is in the up left or the down right or the up right scanset is also a candidate scanline. □

Lemma 4.2.13 *If there exists down left scanset for a given line segment then its up left scanset also exists. The converse of this statement is also true. And similar assertions also hold for the down right and up right scansets.*

Proof: Since there exists down left scanset of a given line segment, there is at least one vertex which can be separated from the given vertex so that it will be in the left subgraph. And this is the condition for the up left scanset to exist. The converse statement and the similar assertions can be easily proved in this way. □

Lemma 4.2.14 If *ith line segment intersects* $(i + 1)$*th line segment then the down right scanset of ith line segment is a subset of the down right scanset of* $(i + 1)$*th line segment. Similar assertions hold for down left, up left and up right scansets.*

Proof: From Definition 4.2.11, notice that finding the down right scanset of ith line segment, we need a minimum k satisfying $i \le k < n$ and $\pi^{-1}(k + 1) > j$ where $\pi^{-1}(i) = j$. If ith line segment intersects $(i + 1)$th line segment *i.e.* $\pi^{-1}(i + 1) < \pi^{-1}(i)$ then the same k will be found for both ith and $(i + 1)$th line segment. Again, for any l, if $\pi(l) > i + 1$ then $\pi(l) > i$ and if $\pi(l - 1) \le i$ then $\pi(l - 1) \le i + 1$. Hence, all scanlines in the down right scanset of ith line segment are also in the down right scanset of $(i + 1)$th line segment. Similarly the other assertions can be proved. □

Lemma 4.2.15 *There are* $O(n)$ *unique scanlines in the collection of down right scansets of all line segment. Similar assertions hold for down left, up left and up right scanlines.*

Proof: We shall prove the first assertion. According to Lemma 4.2.14, if we take the union of the down right scansets of all ith line segment that does not intersect $(i + 1)$th line segment, then we have the collection mentioned in the assertion.

The proof is by induction on n, the number of line segments in the permutation diagram. The assertion is true for $n = 1$ because there is no scanline. Let it holds for $n - 1$. We know that, starting from the permutations of $n - 1$ numbers and placing n to the left or right of all numbers or in between each pair of numbers in consecutive positions would generate the permutations of n numbers. In any of these cases, at most one new scanline will be added in the down right scanset of kth line segment where k is the maximum j such that jth line segment does not intersect the newly added nth line segement. Clearly, such k may not be found. Therefore, the number of scalines in the collection is increased at most by one. Hence, the assertion also holds for n. The similar assertions can also be proved in the same technique. □

Lemma 4.2.16 *If* (k_d, l_d) *is in the down left scanset of ith line segment then scanline* $(k_d - 1, l_d + 1)$ *is in the up left scanset of the same line segment and these two are equivalent. Converse of this assertion also holds. Similarly, if* (k_d, l_d) *is in the down right scanset of* *ith line segment then scanline $(k_d + 1, l_d - 1)$ is in the up right scanset of the same line segment. Converse of this assertion also holds.*

Proof: We shall prove the first assertion. From Definition 4.2.11, we can easily verify that scanline $(k_d - 1, l_d + 1)$ l_d satisfies the conditions of up left scanset. Let. $l_u = l_d + 1$ and $k_d = k_u + 1$. Furthermore $k_u \neq \pi(l_u)$ and $k_d \neq \pi(l_d)$ since (k_u, l_u) and (k_d, l_d) are scanlines, not line segments. We want to show that scanlines (k_u, l_u) and (k_d, l_d) are equivalent. For any j, if $j \leq k_d$ and $\pi^{-1}(j) \leq l_d$ then $j \leq k_u$ and $\pi^{-1}(j) \leq l_u$. Similarly, for any j, if $k_d \leq j$ and $l_d \leq \pi^{-1}(j)$ then $k_u \leq j$ and $l_u \leq \pi^{-1}(j)$. So, the set of line segments that are to the left (or right) of (k_d, l_d) is equal to the set of line segments that are to the left (or right) of (k_u, l_u). Hence, we can say that the remaining line segments that intersect the scanline (k_d, l_d) also intersect the scanline (k_u, l_u) and so the separators found from them are equal. Therefore, from Definition 4.2.5 we can conclude that (k_d, l_d) and (k_u, l_u) are equivalent scanlines. Similarly, we can prove other assertions. □

Lemma 4.2.17 *If we take all four types of scanlines associated with all the line segments then there will be $O(n)$ candidate scanlines. There is no candidate scanlines other than these.*

Proof: The first portion of this lemma is a straightforward implication of Lemma 4.2.15. Lemma 4.2.12 along with its proof shows that we have considered all possible scanlines implicitly and explicitly and only those four types of scanlines are found to be candidate scanlines. □

Corollary 4.2.18 *The number of minimal separators of a permutation graph with n vertices is $O(n)$.*

4.3 Pieces and Candidate Pieces

Now, we are going to define the main concept of designing the vertex-ranking algorithm for permutation graphs using the permutation diagram via minimal separators and candidate scanlines. Since we need optimal ranking, we must satisfy the optimality principle, which says that for a optimal solution each part of the solution itself will be optimal. In case of optimal vertex-ranking of a permutation graph, we need to ensure that each subgraph leading to the optimal ranking of the whole graph is also optimally ranked. At this point, we need some mechanism to represent a subgraph in the permutation diagram. Pieces obviously support this necessity.

Definition 4.3.1 The piece $P = P(s_l, s_r)$ where s_l is to the left of s_r in the permutation diagram is defined by the scanlines s_l and s_r and all the line segments in between the two scanlines. We denote by $V(P)$ the set of vertices in P and by $|P(s_l, s_r)|$ the number of line segments in the piece. $P[U]$ denotes the subpiece of P induced by $U \subset V(P)$. Each piece represents some subgraph of the permutation graph.

Corollary 4.3.2 *The number of pieces in a permutation graph is bounded by n^4 where n is the number of vertices in the graph.*

For computing the vertex-ranking number of a given graph G, we want to use Procedure 3.3.5 and Theorem 3.3.6. We shall compute the vertex-ranking number of all the pieces using dynamic programming technique.

Definition 4.3.3 A scanline s is a *cutting scanline* for $P(s_l, s_r)$ if s is in between s_1 and s_2.

Lemma 4.3.4 [DKKM94] *Let S be a minimal separator of $P(s_l, s_r)$. Then, there is a cutting scanline s for P such that S is exactly the set of those line segments of P crossing s.*

Proof: Consider the permutation diagram of $P(s_l, s_r)$. Let G_l and G_r be subgraphs of $P[V(P) \setminus S]$ such that, every vertex $v \in S$ has a neighbor in G_l and a neighbor in G_r. Then there is a scanline s such that, it does not cross any line segment corresponding to a vertex of $V(P) \setminus S$ and s is in between G_l and G_r *i.e.*, s is in between any two line segment i and j, where i is in G_l and j is in G_r. Thus s is a cutting scanline and S is exactly the set of line segments crossing s. □

Corollary 4.3.5 *Let s be a cutting scanline of a piece $P(s_l, s_r)$. Then $|P(s_l, s_r)| - |P(s_l, s)| - |P(s, s_r)| = |S \setminus (S_l \cup S_r)|$, because both the expressions find the number of line segments in P which cross s.*

Theorem 4.3.6 [DKKM94] *Let $P(s_l, s_r)$ be a piece of a permutation graph $G = \langle V, E\rangle$ and let P have at least one cutting scanline. Further more, $k = max\{r_c(P(s_l, s)), r_c(P(s, s_r))\}$, $c_k = c_{k,P(s_l,s)} + c_{k,P(s,s_r)}$, $S' = S \setminus (S_l \cup S_r)$, $k' = \lceil(|S| - c + c_k)/c\rceil$ if $|S'| + c_k > c$, $k' = \lceil|S|/c\rceil$ if $c_k > c$, othersise $k' = 0$, where S is a minimal separator in P and $c_{k,p}$ is frequency of rank k in P. Then, $r_c(G) = min_S(k + k')$, where the minimum is taken over all the candidate and cutting scanlines s for P.*

Proof: This theorem directly follows from Theorem 3.3.6, Definition 4.3.1, Definition 4.3.2 and Lemma 4.3.4. □

Definition 4.3.7 A piece $P(s_l, s_r)$ is a candidate piece if s_l and s_r are either candidate scanlines or *boundary scanlines*. The scanline (1, 1) and (n, n) are the two boundary scanlines in a permutation diagram.

Corollary 4.3.8 *Number of candidate pieces of a permutation graph is $O(n^2)$, where n is the number of vertices of the graph.*

Since we are considering only minimal separators, it is sufficient to consider only the candidate scanlines and candidate pieces.

Observation 4.3.9 The candidate scanline (i, j)s will be sorted from left to right if sorted in ascending order of $i + j$ and i,

Definition 4.3.10 A piece $P(s_l, s_r)$ is a *fundamental candidate piece* if it has no cutting candidate scanline.

Lemma 4.3.11 *The subgraph induced by the set of vertices in a fundamental candidate piece is always completely connected.*

Proof: Say, the assertion is not true. So, there is at least a pair of vertices, which are not adjacent. Clearly, there is a separator for this pair and hence, a minimal separator, too, exists. Lemma 4.2.9 says that, for this minimal separator there must be a candidate scanline, which will also be a cutting scanline for the given piece. Notice that, this candidate scanline is not only excluded according to Lemma 4.2.17 but also indicates that the given candidate piece will not be fundamental if it is further included. Clearly, these are contradictions. □

Chapter 5 Vertex-Ranking of Permutation Graphs

Section 5.1 describes a sequential algorithm that solves the *c*-vertex-ranking problem for permutation graphs having time complexity $O(n^3)$. Section 5.2 presents the parallel algorithm for the same problem based on the sequential algorithm described in Section 5.1.

5.1 *c*-Vertex-Ranking of Permutation Graphs: Sequential Algorithm

In this section, we describe an efficient sequential algorithm having time complexity $O(n^3)$ to compute the optimal *c*-vertex-ranking number of a permutation graph and to rank its vertices accordingly. Here, almost the same approach is used as in [DKKM94] and [NK99]. The algorithm in [DKKM94] solves the ordinary vertex-ranking problem (*i.e.* $c = 1$) for permutation graphs in $O(n^6)$ time using all the scanlines regardless of it being separating, nonseparating or candidate scanlines and the algorithm in [NK99] solves in $O(n^3)$ time using just the candidate scanlines since only they correspond to the minimal separators.

Algorithm 5.1.1 This algorithm has two procedures– PERM_RANK is the main procedure that computes the optimal *c*-vertex-ranking number and procedure RANK assigns ranks to each vertex of the graph.

```
procedure PERM_RANK(c, n, Perm, Ranks, MaxR, MaxC)
// n, c – number of vertices in the permutation graph and value of c respectively.
// Perm[1: n] – given permutation.
// Ranks[1: n] – output ranks of the vertices.
// MaxR – output vertex-ranking number.
// MaxC – output count of the maximum rank used.
// InvPerm[1: n] – the inverse of the given permutation.
// CScanLine[1:4n] – the sorted list of candidate scanlines.
// NCScanLine – number of candidate scanlines.
//Separator[1:4n, 1:n] – the separator associated with the candidate scanlines.
// CPiece[1:4n, 1:4n] – size of the candidate pieces.
// MaxRank[1: n, 1: n] – maximum ranks of  the pieces.
// MaxCount[1: n, 1: n] – frequencies of max ranks of the pieces.
// Splitter[1: n, 1: n] – the cutting scanline which  led to the optimal ranking of the piece.
// MR, MC, S, i, j, k, l – temporary integer variables.

1. // Compute the Inverse Permutation InvPerm of the given permutation.
   for $1 \le i \le n$ do
           $InvPerm[Perm[i]] = i$
2. // Find all the four types of scanlines associated with all line segment.
   // From equivalent pairs take those having less top index.
   // Take also scanlines (1, 1) and (n, n).
   for $1 \le i \le n$ do
      begin
           $j = InvPerm[i]$
           // Down Left Scanset
           Find maximum $k$ such that $i \ge k > 1$ and $InvPerm[k-1] < j$.
           for $j - 1 \ge l \ge InvPerm[k-1]$
                if $Perm[l] < i$ and $Perm[l+1] \ge i$ then
                     $(k, l)$ is in the down left scanset of ith line segment
                endif.
           // Up Left Scanset
           Find maximum $l$ such that $j \ge l > 1$ and $Perm[l-1] < i$.
           for $i - 1 \ge k \ge l - 1$
                if $InvPerm[k] < j$ and $InvPerm[k+1] \ge j$ then
                     $(k, l)$ is in the up left scanset of ith line segment
                endif.

           Take the up left scanset only, because any scanline in the down left scan
                set is equivalent to some scanline in the up left scan set.
```

```
            // Down Right Scanset
            Find minimum $k$ such that $i \le k < n$ and $InvPerm[k + 1] > j$.
            for $j + 1 \le l \le InvPerm[k + 1]$
                if $Perm[l] > i$ and $Perm[l - 1] \le i$ then
                    $(k, l)$ is in the down right scanset of $i$th line segment
                endif.
            // Up Right Scanset
            Find minimum $l$ such that $j \le l < n$ and $Perm[l + 1] > i$.
            for $i + 1 \le k \le l + 1$
                if $InvPerm[k] > j$ and $InvPerm[k - 1] \le j$ then
                    $(k, l)$ is in the up right scanset of $i$th line segment
                endif.
            Take the down right scanset only, because any scanline in the up right scan
                set is equivalent to some scanline in the down right scan set.
        end
    Take also scanlines - $(1, 1)$ and $(n, n)$.
```

3. Sort all the scanlines found from Step 2 in ascending order of (top + bottom) index. In case of equality, use ascending order of the top index. In the sorted list, the duplicate scanlines are in consecutive position. Eliminate them. The remaining are all unique scanlines.

   NCScanLine = number of the unique candidate scanlines.

   CScanLine(k) is the kth candidate scanline where $1 \le k \le NCScanLine$.

4. // Find the separators associated with each of the scanlines.

```
    for $1 \le k \le NCScanLine$ do
        for $1 \le j \le n$ do
            if $CScanLine[k]$ intersects $j$th line segment then
                $Separator[k, j] = 1$
            else
                $Separator[k, j] = 0$
            endif
```

5. // Find the size of the pieces

```
    for $1 \le i \le NCScanLine - 1$ do
    for $i + 1 \le j \le NCScanLine$ do
        begin
            $CPiece[i, j] = 0$
            for $1 \le k \le n$ do
                if kth line segment is in between $CScanLine[i]$ and $CScanLine[j]$ then
                    $CPiece[i, j] = CPiece[i, j] + 1$
                endif
        end.
6. // Initialize the first diagonal of the table used in dynamic programming.
   for $1 \le k \le NCScanLine - 1$ do
       begin
           $MaxRank[k, k + 1] = \text{ceil}(CPiece[k, k + 1] / c)$
           $MaxCount[k, k + 1] = CPiece[k, k + 1] \bmod c$
           $Splitter[k, k + 1] = \text{NULL};$
       end.
7. // Compute vertex-ranking numbers using dynamic programming
   for $2 \le l \le NScanLine - 1$ do
       for $1 \le i \le NCScanLine - l$ do
           begin
               $k = i + l$
               $MaxRank[i, k] = \text{ceil}(CPiece[i, k] / c)$
               $MaxCount[i, k] = CPiece[i, k] \bmod c$
               $Splitter[i, k] = \text{NULL}$
               for $i + 1 \le j \le k - 1$
                   begin
                       $MR = \max(MaxRank[i, j], MaxRank[j, k])$
                       $MC = 0$
                       if $MR = MaxRank[i, j]$ then
                           $MC = MC + MaxCount[i, j]$
                       endif
                       if $MR = MaxRank[j, k]$ then
                           $MC = MC + MaxCount[j, k]$
                       endif
                    $S = CPiece[i, k] - CPiece[i, j] - CPiece[j, k]$
                    if $MC > c$ then
                        $MR = MR + \text{ceil}(S / c)$
                        $MC = S \bmod c$
                    elseif $S + MC > c$ then
                        $MR = MR + \text{ceil}((S + MC - c) / c)$
                        $MC = (S + MC - c) \bmod c$
                    else
                        $MC = MC + S$
                    endif
                    if $MaxRank[i, k] > MR$ or
                            ($MaxRank[i, k] = MR$ and $MaxCount[i, k] > MC$) then
                      $MaxRank[i, k] = MR$
                      $MaxCount[i, k] = MC$
                      $Splitter[i, k] = j$;
                  endif
              end
          end
8.  call RANK($Ranks$, 1, $Splitter[1, NCScanLine]$, $NCScanLine$)
    $MaxR = MaxRank[1, NCScanLine]$
    $MaxC = MaxCount[1, NCScanLine]$
end PERM_RANK

procedure RANK($Ranks$, $i, j, k$)
    if $i = k + 1$ then
        Rank first $c$ vertices whose line segments lie in between scanlines
            $CScanLine[i]$ and $CScanLine[k]$ with rank 1, then next such $c$ vertices with
            rank 2, and so on;
        return
    endif
    if $j$ is NULL then
        Rank first $MaxCount[i, k]$ vertices whose line segments lie in between scanlines
            $CScanLine[i]$ and $CScanLine[k]$ with rank $MaxRank[i, k]$, then next such $c$
            vertices with $MaxRank[i, k] - 1$, and so on.
            return
        endif;
        call RANK(Ranks, i, Splitter[i,j], j);
        call RANK(Ranks, j, Splitter[j, k], k);
        Rank first MaxCount[i, k] vertices of Separtor[j] whose line segments lie in between
            scanlines CScanLine[i] and CScanLine[k] with ranks MaxRank[i, k], next such c
            vertices with MaxRank[i, k] – 1, and so on.
    return;
end RANK
```

The complexity analysis of Algorithm 5.1.1 is shown in the following theorem.

Theorem 5.1.2 *Optimal c-vertex-ranking of a permutation graph can be found in $O(n^3)$ sequential time.*

Proof: Let us analyze step wise. Step 1 needs $O(n)$ time. Step 2 needs $O(n^2)$ time since finding DLS(i), ULS(i), DRS(i), URS(i) each need $O(n)$ time. Step 3 needs $O(n^2)$ time since number of scanlines found in Step 2 is $O(n)$. Step 4 needs $O(n^2)$ time. Step 5 has a dominating time complexity of $O(n^3)$ while Step 6 takes just $O(n)$ time. Step 7 is clearly another most time consuming part of the algorithm and it runs in $O(n^3)$ time. At last Step 8 assigns the ranks to the vertices recursively in $O(n^2)$ time because determining the line segments in between two given scanlines requires $O(n)$ time. So, it is obvious that the sequential time complexity of Algorithm 5.1.1 is $O(n^3)$.

5.2 *c*-Vertex-Ranking of Permutation Graphs: Parallel Algorithm

In this section, we describe a parallel algorithm based on the sequential algorithm presented in Section 5.1 that computes the optimal *c*-vertex-ranking number of a permutation graph and to rank its vertices accordingly. This algorithm runs in $O(\log^2 n)$ parallel time using $O(n^3 \log n)$ operations on the CREW PRAM.

Step 7 of Algorithm 5.1.1 is the most critical part from the point of view of parallel implementation. Straightforward implementation of this step would result into non-polylogrithmic complexity. In this step a two-dimensional table as shown in Figure 5.1 is used. If s_i, s_j are the ith and jth scanline in the sorted list of nonequivalent candidate scanlines found in Step 3 of Algorithm 5.1.1, then cell(i, j) contains the maximum rank used, count of the maximum rank and the splitter of this ranking of the piece $P(s_i, s_j)$. Cells in each diagonal can be computed in parallel whereas cells in different diagonals can not be.

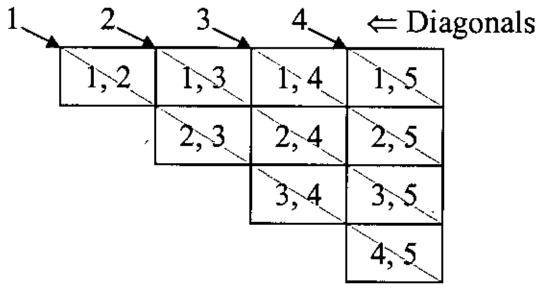


Figure 5.1 Table T used in Step 7 of Algorithm 5.1.1

Let us first define a graph that resembles the appearance of the above table. Then we map the dynamic programming problems into shortest path problems.

Definition 5.2.1 D = $\langle V, E\rangle$ is a directed and weighted graph defined as follows:

The set of vertices $V = \{(i,j)\colon 1 \le i < j \le n\} \cup \{(0, 0)\}$

The set of edges $E = \{\langle (i,j) \rightarrow (i,k)\rangle : 1 \le i < j < k \le n\} \cup$

$$\{\langle (i,j) \rightarrow (k,j)\rangle : 1 \le k < i < j \le n\} \cup$$

$$\{\langle (0,0) \rightarrow (i,i+1)\rangle : 1 \le i < n\}$$

The cost function $w(\langle (i,j) \rightarrow (i,k)\rangle) = \langle R(j,k), C(j,k), S(i,j,k)\rangle$ if $1 \le i < j < k \le n$,

$$w(\langle (i,j) \rightarrow (k,j)\rangle) = \langle R(i,k), C(i,k), S(k,i,j)\rangle \text{ if } 1 \le k < i < j \le n,$$

$$w(\langle (0,0) \rightarrow (i,i+1)\rangle) = \langle R(i,i+1), C(i,i+1), 0\rangle \text{ if } 1 \le i < n$$

where s_k is the kth scanline,

$R(i,j) = r_c(P(s_i, s_j))$,

$C(i,j)$ = count of the rank of $R(i,j)$ in $P(s_i, s_j)$

$S(i,j,k) = |P(s_i, s_k)| - |P(s_i, s_j)| - |P(s_j, s_k)|$

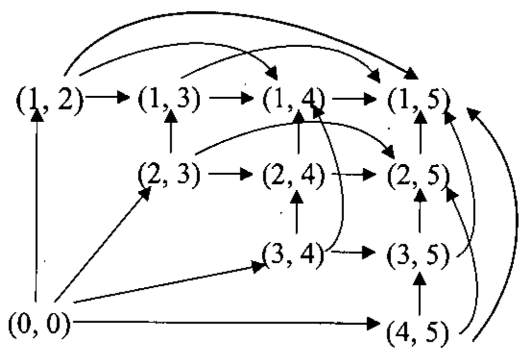


Figure 5.2: Graph D with $n = 4$.

Definition 5.2.2 Let the shortest path estimates in D from vertex (i, j) to (k, l) is $W(\langle(i,j) \Rightarrow (k,l)\rangle) = \langle R,C,S,P\rangle$, from vertex (i, j) to (p, q) is $W(\langle(i,j) \Rightarrow (p,q)\rangle) = \langle R_1,C_1,S_1,P_1\rangle$ and from vertex (p, q) to (k, l) costs $W(\langle(p,q) \Rightarrow (k,l)\rangle) = \langle R_2,C_2,S_2,P_2\rangle$ where R is the maximum rank used, C is its count, S is the number of scanlines in the separator set and P is the predecessor. Then, executing the following program segment that updates the estimate $W(\langle(i,j) \Rightarrow (k,l)\rangle)$ is called *path relaxation*.

```
$R_3 = max(R_1, R_2)$                          // Compute the maximum
$C_3 = 0$
If $R_3 = R_1$ then $C_3 = C_3 + C_1$ endif     // Compute the count of the maximum rank
If $R_3 = R_2$ then $C_3 = C_3 + C_2$ endif
If $C_3 > c$ then                              // Compute additional ranks need for S
    $R_3 = R_3 + ceil(S_2 / c)$
    $C_2 = S_2 \bmod c$
elseif $S_2 + C_3 > c$ then
    $R_3 = R_3 + ceil((S_2 + C_3 - c) / c)$
    $C_3 = (S_2 + C_3 - c) \bmod c$
else
    $C_3 = C_3 + S_2$
endif
If ($R > R_3$ or ($R = R_3$ and $C > C_3$)) then    //Relax this with previous estimate
    $R = R_3$, $C = C_3$, $S = S_1$, $P = P_2$
endif
```

Definition 5.2.3 *Length* of edges $\langle (i,j) \to (i,k) \rangle$, $\langle (j,k) \to (i,k) \rangle$ and $\langle (0,0) \to (i,i+1) \rangle$ in D are $(k-j)$, $(j-i)$ and 1 respectively. A vertex (k, l) is $(l-k)$ *unit apart* from vertex (0, 0) and $|(i-k)| + |(j-l)|$ *unit apart* from vertex (i, j). A vertex (k, l) is on $(l-k)$th *diagonal*. There are $k+1$ vertices on kth diagonal. A vertex $(i, i+k)$ on kth diagonal is $|k-l|$ *diagonal apart* from another vertex $(j, j+l)$ on lth diagonal and there are $|k-l|+1$ vertices on lth diagonal that are $|k-l|$ unit apart from $(i, i+k)$.

Theorem 5.2.4 [B92] *Finding a shortest path from* (0, 0) *to* $(1, n)$ *in D computes Cell*$(1, n)$ *of the table T used in Step 7 of Algorithm 5.1.1 where n is the number of scanlines found in Step 3 of the same and path relaxation is as defined in Definition 5.2.2.*

Proof: The proof is by induction on diagonals of T. For $1 \leq i < n$, each vertex $(i, i+1)$ is on the first diagonal and the shortest path from vertex (0, 0) is $w(\langle (0,0) \to (i,i+1) \rangle)$. This is the initialization of the first diagonal of the table T. We know that, a vertex (i, j) in D corresponds to Cell(i, j) in T. Assume that, shortest paths from (0, 0) to all vertices (i, j) which are on any of the first $(k-1)$th diagonals are available. From the defintion of cost function as in Definition 5.2.1, all edges of length up to $k-1$ are available. Relaxing all the incoming edges, the shortest path from (0, 0) to the vertices (i, j), which are on kth diagonal can be computed in the same fashion that is used in computation of table cells. So, by induction the assertion holds. □

Construction of the graph D is not straightforward. Weight of the edges of length greater than one cannot be available because the R and C component of the weight function is dependent on shortest path length of some other vertex from (0, 0). Hence, the edges are constructed and relaxed whenever the weight function is available.

Theorem 5.2.5 [B92] *If a shortest path from* (0, 0) *to* (i, k) *in D contains an edge* $\langle (i,j) \to (i,k) \rangle$ *then there is a dual shortest path containing edge* $\langle (j,k) \to (i,k) \rangle$. *The converse is also true.*

Proof: Let the edge $\langle (i,j) \to (i,k) \rangle$ is in a shortest path from (0, 0) to (i, k). From Definition 5.2.2, $R(i, k)$ = relaxation of the shortest path $\langle (0,0) \Rightarrow (i,j) \rangle$ and the edge

$\langle (i,j) \rightarrow (i,k) \rangle$ = max(R(i, j), R(j, k)) + f($C(i, j)$, $C(j, k)$, $S(i, j, k)$) where f is a function that computes additional ranks used for S. Similarly, if the edge $\langle (j,k) \rightarrow (i,k) \rangle$ is in a shortest path from (0, 0) to (i, k) then $R(i, k)$ = relaxation of the shortest path $\langle (0,0) \Rightarrow (j,k) \rangle$ and the edge $\langle (j,k) \rightarrow (i,k) \rangle$ = max(R(i, j), R(j, k)) + f($C(i, j)$, $C(j, k)$, $S(i, j, k)$). So, we can compute the shortest path from (0, 0) to (i, k) in either way. □

Lemma 5.2.6 [B92] *For all vertices (i, k) in D, R(i, k) and C(i, k) can be computed by a path having edges of length no longer than* $\lceil (k-i)/2 \rceil$.

Proof: The proof is by induction on edge length. According to Theorem 5.2.5, if the edge $\langle (i,j) \rightarrow (i,k) \rangle$ is in a shortest path $\langle (0,0) \Rightarrow (i,k) \rangle$ then the edge $\langle (j,k) \rightarrow (i,k) \rangle$ is also in a shortest path $\langle (0,0) \Rightarrow (i,k) \rangle$. The length of the edges $\langle (i,j) \rightarrow (i,k) \rangle$ and $\langle (j,k) \rightarrow (i,k) \rangle$ are $k - j$ and $j - i$ respectively. If the edge $\langle (i,j) \rightarrow (i,k) \rangle$ is longer than $\lceil (k-i)/2 \rceil$ *i.e.* $(k - j) > \lceil (k-i)/2 \rceil$ then $(j - i) \leq \lceil (k-i)/2 \rceil$ because $(k - j) + (j - i) = (k - i)$ implying edge $\langle (j,k) \rightarrow (i,k) \rangle$ is no longer than $\lceil (k-i)/2 \rceil$. Again, by induction, a shortest path $\langle (0,0) \Rightarrow (j,k) \rangle$ cannot contain any edge of length greater than $\lceil (k-j)/2 \rceil$. Hence, the shortest path can be computed using the smaller edge. □

Lemma 5.2.7: *Assume all shortest paths have been computed between each pair of vertices up to* 2^{r-1} *unit apart. Then, one (min, max) matrix multiplication computes the shortest path for all pairs of vertices up to* 2^r *unit apart.*

Proof: All shortest paths between each pair of vertices up to 2^{r-1} unit apart are available. This includes the shortest paths from (0, 0) to the vertices up to the (2^{r-1})th diagonal because a vertex on (2^{r-1})th dignonal is (2^{r-1}) unit apart from (0, 0). Using these values every edge of length 2^{r-1} or smaller can be computed. By Lemma 5.2.6, placing and relaxing these edges in D and then performing one (min, max) matrix multiplication computes the shortest paths from (0, 0) to all vertices up to 2^rth diagonal. □

Lemma 5.2.8 *There are $O(d)$ vertices that are $d/2$ distances apart from both the vertices of a given pair of vertices and finalizing the shortest path between this pair requires $O(d)$ path relaxations.*

Proof: According to Lemma 5.2.7, computing the shortest path between two vertices 2^r unit apart requires the shortest path of pairs up to 2^{r-1} unit apart. Form the graph D, observe that for any pair of vertices 2^r unit apart there are $O(2^{r-1})$ vertices which are 2^{r-1} unit apart from both of them. Path relaxation cannot be done via a vertex that is less than 2^{r-1} units apart from either vertex and hence more than 2^{r-1} units apart from the other vertex. So to finalize the shortest path between a given pair the path relaxation is done $O(2^{r-1})$ times. In fact, by Lemma 5.2.6, this implication can easily be generalized for shortest path of any pair of vertices d unit apart. □

Lemma 5.2.9 *Computing the shortest path from $(0, 0)$ to $(1, n)$ requires $O(log^2n)$ parallel time on the CREW PRAM.*

Proof: According to Lemma 5.2.7, $O(\log n)$ iterations is needed to compute the shortest path from $(0, 0)$ to $(1, n)$ and by Lemma 5.2.8, it needs $O(n)$ path relaxations taking $O(\log n)$ time. So, the time complexity becomes $O(\log^2 n)$. CREW PRAM is needed because in any iteration, shortest paths computed in previous iterations are concurrently read but finally only one processor writes the shortest path of a pair. □

Lemma 5.2.10 *Computing the shortest path from* $(0, 0)$ *to* $(1, n)$ *where* $n = 2^r + 1$ *requires computing shortest path of some other $O(n^2 logn)$ pair of vertices in D.*

Proof: According to Lemma 5.2.7, computing the shortest paths between a pair 2^r diagonal apart requires shortest paths of the vertices 2^{r-1} diagonal apart. Like divide and conquer strategy, these shortest paths can also be broken down into shortest paths of the vertices 2^{r-2} diagonal apart, and so on. But the shortest paths will be computed in a bottom up fashion. In general, in iteration l, we compute for all $0 \leq k \leq (n / 2^l)$, shortest paths from each of the $(k2^{l-1} + 1)$ vertices on $(k2^{l-1})$th diagonal to all the $(2^{l-1} + 1)$ vertices that are 2^{l-1} unit apart from it and on $((k + 1)2^{l-1})$th diagonal. There are $\log n$

iterations. Hence, shortest paths of $\sum_{0 \le l \le \log n} \sum_{0 \le k \le \lceil n/2^l \rceil} (k2^{l-1}+1)(2^{l-1}+1) =$ $O(n^2 \log n)$ pair of vertices need to be computed in total. □

Lemma 5.2.11 *Computing the shortest path from* (0, 0) *to* (1, n) *where* $n = 2^r + 1$ *costs* $O(n^3 \log n)$ *operations.*

Proof: According to Lemma 5.2.10, shortest paths of $O(n^2 \log n)$ pair of vertices need to be computed and for each such pair requires $O(n)$ operations by Lemma 5.2.8. So, the operation complexity becomes $O(n^3 \log n)$. □

Remark 5.2.12 If $n \neq 2^r + 1$ then we can add some additional vertices along with edges of cost $\langle 0,0,0 \rangle$ in the graph so that for the new graph n become equal to $2^r + 1$.

Algorithm 5.2.13 This algorithm is actually the parallel implementation of Algorithm 5.1.1 and hence a stepwise direct correspondence is easily revealed.

```
procedure PERM_RANK(c, n, Perm, Ranks, MaxR, MaxC)
// n, c – number of vertices in the permutation graph and value of c respectively.
// Perm[1: n] – given permutation.
// Ranks[1:n] – output ranks of the vertices.
// MaxR – output vertex-ranking number.
// MaxC – output count of the maximum rank used.
// InvPerm[1:n] – the inverse of the given permutation.
// CScanLine[1:4n] – the sorted list of candidate scanlines.
// NCScanLine – number of candidate scanlines.
//Separator[1:4n, 1:n] – the separator associated with the candidate scanlines.
// CPiece[1:4n, 1:4n] – size of the candidate pieces.
// R[0:n, 0:n, 0:n, 0:n] – maximum rank component of W.
// C[0:n, 0:n, 0:n, 0:n] – rank count component of W.
// S[0:n, 0:n, 0:n, 0:n] – separator component of W.
// P[0:n, 0:n, 0:n, 0:n] – predecessor component of W.
// Piece[1:n], MR, Sep, MC, i, j, k, l – temporary variables.
```

```
1.  // Compute the Inverse Permutation InvPerm of the given permutation.
    for $1 \le i \le n$ pardo
            $InvPerm[Perm[i]] = i$
2.  // Find all the four types of scanlines associated with all linesegments.
    // From equivalent pairs take those having less top index.
    // Take also scanlines (1, 1) and (n, n).
    for $1 \le i \le n$ pardo
        begin
            $j = InvPerm[i]$
            // Down Left Scanset
            Find maximum $k$ such that $i \ge k > 1$ and $InvPerm[k-1] < j$.
            for $j-1 \ge l \ge InvPerm[k-1]$ pardo
                if $Perm[l] < i$ and $Perm[l+1] \ge i$ then
                    $(k, l)$ is in the down left scanset of $i$th line segment
                endif.
            // Up Left Scanset
            Find maximum $l$ such that $j \ge l > 1$ and $Perm[l-1] < i$.
            for $i-1 \ge k \ge l-1$ pardo
                if $InvPerm[k] < j$ and $InvPerm[k+1] \ge j$ then
                    $(k, l)$ is in the up left scanset of $i$th line segment
                endif.
            Take the up left scanset only because any scanline in the down left scan set
                is equivalent to some scanline in the up left scan set.
            // Down Right Scanset
            Find minimum $k$ such that $i \le k < n$ and $InvPerm[k+1] > j$.
            for $j+1 \le l \le InvPerm[k+1]$ pardo
                if $Perm[l] > i$ and $Perm[l-1] \le i$ then
                    $(k, l)$ is in the down right scanset of $i$th line segment
                endif.
            // Up Right Scanset
            Find minimum $l$ such that $j \le l < n$ and $Perm[l+1] > i$.
            for $i+1 \le k \le l+1$ pardo
```

```
                if $InvPerm[k] > j$ and $InvPerm[k-1] \le j$ then
                        $(k, l)$ is in the up right scanset of $i$th line segment
                endif.
            Take the down right scanset only because any scanline in the up right scan
                set is equivalent to some scanline in the down right scan set.
    end
```

Take also scanlines - (1, 1) and (n, n).

3. Using some parallel sorting algorithm sort all the scanlines found from Step 2 in ascending order of (top + bottom) index. In case of equality, use ascending order of the top index. In the sorted list, the duplicate scanlines are in consecutive position. Eliminate them. The remaining are all unique scanlines. Using parallel list ranking algorithm compute the indices of the scanlines.

   NCScanLine = number of the unique candidate scanlines.

   CScanLine(k) is the kth candidate scanline where $1 \le k \le NCScanLine$.

4. // Find the separators associated with each of the scanlines.

```
for $1 \le k \le NCScanLine$ pardo
    for $1 \le j \le n$ pardo
        if $CScanLine[k]$ intersects $j$th line segment then
            $Separator[k, j] = 1$
        else
            $Separator[k, j] = 0$
        endif
```

5. // Find the size of the pieces

```
for $1 \le i \le NCScanLine - 1$ pardo
    for $i + 1 \le j \le NCScanLine$ pardo
        $CPiece[i, j]$ = Sum of $Piece[k]$s where $1 \le k \le n$ and
            $Piece[k] = 1$ if $k$th line segment is in between $CScanLine[i]$ and
            $CScanLine[j]$; otherwise $Piece[k] = 0$
```

6. // Initialize shortest paths of the vertices in the first diagonals from (0, 0).

```
for $1 \le k \le NCScanLine - 1$ pardo
    begin
        $R[0, 0, k, k+1]$ = ceil($CPiece[k, k+1] / c$)
        $C[0, 0, k, k+1] = CPiece[k, k+1] \bmod c$
```

```
            $S[0, 0, k, k + 1]$ = NULL;
            $P[0, 0, k, k + 1] = (0, 0)$;
        end.
7.  // Compute the shortest paths.
    for $l = 0$ to $\log_2 n$ do
        begin
            compute the cost of the edges of length up to $2^l$ and take the minimum of the
                cost of this edge and the already computed shortest path cost between
                the vertices on which the edge is incident.
            for $0 \le k \le (n / 2^l)$ pardo
              for each vertex $(i, j)$ on $(k2^{l-1})$th diagonal pardo
                for each vertex $(k, l)$ on $((k + 1)2^{l-1})$th diagonal and $2^{l-1}$ unit apart
                    from vertex $(i, j)$ pardo
                  for each vertex $(p, q)$ that is $2^{l-2}$ unit apart from both $(i, j)$ and $(k, l)$ do
                    compute the shortest path from $(i, j)$ to $(k, l)$ through vertex $(p, q)$
                    and relax this with the previously computed value.
        end
```

Theorem 5.2.14 *Optimal c-vertex-ranking of a permutation graph can be found in $O(\log^2 n)$ parallel time using $O(n^3 \log n)$ operations on the CREW PRAM.*

Proof: Let us analyze step wise. Step 1 of Algorithm 5.2.13 need $O(1)$ time $O(n)$ operations on EREW PRAM. Step 2 needs at most $O(1)$ time $O(n^2)$ operations on CREW PRAM since finding DLS(i), ULS(i), DRS(i), URS(i) each need $O(n)$ operations. Step 3 needs at most $O(\log n)$ time $O(n \log n)$ operations for sorting [J92] on CREW PRAM since number of candidate scanlines found in Step 2 is $O(n)$ and needs $O(\log n)$ time and $O(n)$ operations for list ranking [J92]. Step 4 requires $O(1)$ time $O(n^2)$ operations on CREW PRAM. In Step 5, computing the size of the pieces needs $O(\log n)$ time $O(n^3)$ operations in CREW PRAM. Step 6 requires $O(1)$ time and $O(n)$ operations on EREW PRAM. By Lemma 5.2.9 and 5.2.11, Step 7 runs in $O(\log^2 n)$ parallel time using $O(n^3 \log n)$ operations on CREW PRAM. Hence Algorithm 5.2.14 has a overall time complexity $O(\log^2 n)$, operation complexity $O(n^3 \log n)$ on the CREW PRAM model. □

Chapter 6 Conclusion

6.1 Conclusion

Deogun *et al.* have given a sequential algorithm that solves the *c*-vertex-ranking problem with $c = 1$ for permutation graphs in $O(n^6)$ sequential time using scanlines and pieces. Later, we have solved the same problem in $O(n^3)$ sequential time [NK99]. But no parallel algorithm to solve the *c*-vertex-ranking problem for permutation graphs has been reported in the literature. In this thesis, we have devised a parallel algorithm for the same problem with any value of *c* that runs in $O(\log^2 n)$ parallel time using $O(n^3 \log n)$ operations on the CREW PRAM model.

6.2 Guideline For Further Research

In [NK99], we have solved the *c*-vertex-ranking problem for permutation graphs in $O(n^3)$ sequential time. But the parallel algorithm presented in this thesis uses $O(n^3 \log n)$ operations. Therefore, there is a scope to put further reduction on the operation complexity of our algorithm.

References

[B92] P.G. Bradford, Efficient parallel dynamic programming, *Proceedings of The 30th Annual Allerton Conference on Communication, Control and Computing*, 1992, University of Illinois, pp. 185-194.

[DKKM94] J.S. Deogun, T. Kloks, D. Kratsch, H. Müller, On vertex ranking for permutation and other graphs, *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science Springer-Verlag*, 775 (1994), pp. 747-758.

[DR83] I.S. Duff and J.K. Reid, The multifrontal solution of indefinite sparse symmetric linear equations, *ACM Transactions on Mathematical Software* 9 (1983), pp. 302-325.

[IRV88] A.V. Iyer, H.D. Ratliff, and G. Vijayan, Optimal node ranking of trees, *Information Processing Letters,* 28 (1988), pp. 225-229.

[J92] J. Joseph, *An Introuction to Parallel Algorithms*, Addison Wesley, 1992.

[KZN00] M.A. Kashem, X. Zhou, and T. Nishizeki, Algorithms for generalized vertex-ranking of partial *k*-trees, *Theoritical Computer Science*, 240(2000), pp. 407-427.

[K93] T. Kloks, Treewidth, Ph.D. *Thesis*, *Utrecht University*, *The Netherlands*, 1993.

[L80] C.E. Leisertion, Area efficient graph layouts for VLSI, *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science*, 1980, pp. 270-181.

[L90] J.W.H. Liu, The role of elimination trees in sparse factorization, *SIAM Journal of Matrix Analysis and Applications*, 11 (1990), pp. 134-172.

[NK99] M.A.H Newton, M.A. Kashem, An efficient alogirthm for vertex-ranking of permutation graphs, *Proceedings of International Conference on Computer and Information Technology*, *Sylhet*, *Bangladesh*, 1999, pp. 315-320.

[P88] A. Pothen, The Complexity of optimal elimination trees, *Technical Report CS-88-13, Pennsylvania State University, USA*, 1988.

[RK01] M.Z. Rahman, M.A. Kashem, An optimal parallel algorithm for *c*-vertex-ranking of trees, *Proceedings of International Conference on Electrical and Computer Engineering*, *Dhaka*, *Bangladesh*, 2001, pp. 277-280.

[S89] A.A. Schäffer, Optimal node ranking of trees in linear time, *Information Processing Letters*, 33 (1989/90), pp. 309-322.

[SDG92] Sen, H. Deng, and S. Guha, On a graph partition problem with application to VLSI layout, *Information Processing Letters*, 43 (1992), pp. 87-94.

[ZNN95] X. Zhou, H. Nagai, and T. Nishizeki, Generalized vertex-rankings of trees, *Information Processing Letters*, 56(1995), pp. 321-328.