RKPianGraphSort: A Graph based Sorting Algorithm

Rajat K. Pal

Department of Computer Science and Engineering
University of Calcutta
92, A. P. C. Road
Kolkata – 700 009, India
rajatkp@vsnl.net

Abstract – *Sorting is a well-known problem frequently used in many aspects of the world of computational applications. Sorting means arranging a set of records (or a list of keys) in some (increasing or decreasing) order. In this paper, we propose a graph based comparison sorting algorithm, designated as RKPianGraphSort, that takes time* $\Theta(n^2)$ *in the worst-case, where n is the number of records in the given list to be sorted.*

Keywords **–** Sorting, Comparison sort, Record, Satellite data, Graph, Algorithm, Complexity.

1. Introduction

There are several algorithms that solve the following ***sorting problem*** [1, 2, 3, 5, 7]:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (or reordering) $\langle a_1', a_2', \ldots, a_n' \rangle$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.

The input sequence is usually an n-element array, although it may be represented in some other fashion, such as linked list. In practice, the numbers to be sorted are rarely isolated values. Each is usually part of a collection of data called a *record*. Each record contains a *key*, which is

the value to be sorted, and the remainder of the record consists of *satellite data*, which are usually carried around with the key. In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well.

As for example, let us assume that a given unsorted list is as follows.

6 4 1 9 5 8 3 7 2

The subsequent sorted sequence of the elements in this list is given below, in non-decreasing fashion.

1 2 3 4 5 6 7 8 9

Here the satellite information may be to identify whether two keys a_i and a_j in their respective positions i and j in the given list are in order, in computing the sequence in some sorted fashion.

Now there is a lot of sorting algorithms in the present day world. Out of which, *insertion sort* takes $\Theta(n^2)$ time in the worst case. *Merge sort* has a better asymptotic running time, $\Theta(n\lg n)$, in the worst case. *Heapsort* sorts n numbers in $O(n\lg n)$ time, whereas the worst case running time of *quicksort* is $\Theta(n^2)$. The average case running time of quicksort is $\Theta(n\lg n)$, though, it generally outperforms heapsort in practice [3].

Insertion sort, merge sort, heapsort, and quicksort are all comparison sorts: they determine the sorted order of an input array by comparing elements, and it has been proved that heapsort and merge sort are asymptotically optimal comparison sorts [1, 2, 3, 5, 7].

Though $\Omega(n\lg n)$ is a lower bound on the worst case running time of any comparison sort of n inputs, there are a few mechanical sorting algorithms, viz., counting sort, radix sort, bucket sort, etc. that run in linear time, if the size of each of the numbers to be sorted is bounded by a constant [3].

In this paper, we have developed a graph based sorting algorithm, designated as *RKPianGraphSort*, that sorts the elements in the given list, based on their positions in the list, from the graph rather than merely comparing elements as done in different comparison sorting algorithms. The worst-case time complexity of this algorithm is $\Theta(n^2)$, where n is the number of records in the given list to be sorted.

The paper is organized as follows. In Section 2, we formulate the graph based sorting algorithm, and develop it. In Section 3, we compute the computational complexity of the algorithm, and study the stability of the algorithm. In Section 4, we conclude the paper with a few remarks.

2. The Sorting Algorithm

Let $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_n\}$ be the given sequence (or list) of n *unsorted* elements (or keys). The elements are numbered 1 through n, from left to right of the list, to label their positions in Π. That is, Π_1 is the first element in Π, Π_2 is the second element in Π, and so on. The position i for some element Π_i in Π, is important in comparing other elements in their respective positions with Π_i in Π.

In this section, we formulate a graph based sorting algorithm *RKPianGraphSort*, named after *RKP*, developed in this paper. This algorithm sorts the elements in Π in non-decreasing order. In computing the *RKPian graph*, based on the given sequence of n elements in Π, we do the following.

For each of the elements in Π, we introduce a vertex in the graph; hence the graph consists of a total of n isolated vertices. Now for introducing edges to the graph, we do the following. For any pair of elements Π_i and Π_j, that are in positions i and j, respectively, in Π, we

introduce an edge between the corresponding vertices of Π_i and Π_j, only if $(i-j)(\Pi_i-\Pi_j)$ is nonnegative. In other words, we do not introduce any edge between the corresponding vertices in the graph, if they are out of order (i.e., if $\Pi_i > \Pi_j$). This has been explained in Figure 1, for the given unsorted list of nine numbers assumed in the introductory section.

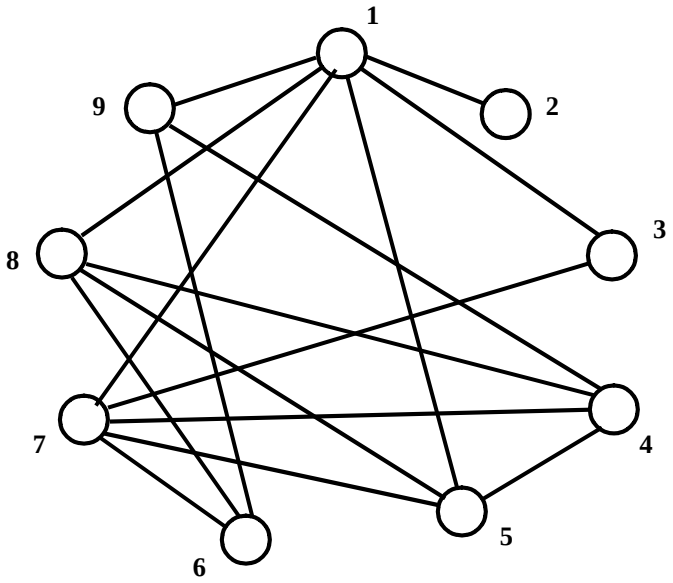

Figure 1. The *RKPian graph* for the given sequence $\Pi = \langle 6\ 4\ 1\ 9\ 5\ 8\ 3\ 7\ 2 \rangle$. An edge $\{v_i,v_j\}$ in this graph indicates that the corresponding elements Π_i and Π_j are in non-descending order in Π.

Note that the computed *RKPian graph* is transitively orientable [4], as stated in the following lemma.

Lemma 1. *For any three vertices v_i, v_j, and v_k in the RKPian graph corresponding to the elements Π_i, Π_j, and Π_k in Π, such that $\Pi_i \leq \Pi_j$ and $\Pi_j \leq \Pi_k$ for their respective positions i, j, and k, where $i < j$ and $j < k$, if the edge $\{v_i,v_j\}$ is oriented from v_i to v_j (i.e., $v_i \rightarrow v_j$) and the edge $\{v_j,v_k\}$ is oriented from v_j to v_k (i.e., $v_j \rightarrow v_k$), then the edge $\{v_i,v_k\}$ must be there in the graph and this edge is oriented from v_i to v_k (i.e., $v_i \rightarrow v_k$).*

Proof. From the given premise we have $\Pi_i \leq \Pi_j$ and $\Pi_j \leq \Pi_k$; therefore, $\Pi_i \leq \Pi_k$. Moreover, $i < j$ and $j < k$ signify that $i < k$. So, the edge $\{v_i,v_k\}$ must be there in the *RKPian graph*. Consequently, if (i) for $\Pi_i \leq \Pi_j$ and $i < j$, the edge between the corresponding vertices v_i and v_j is oriented from v_i to v_j, and (ii) for $\Pi_j \leq \Pi_k$ and $j < k$, the edge between the corresponding vertices v_j and v_k is oriented from v_j to v_k, then for $\Pi_i \leq \Pi_k$ and $i < k$, the edge between the corresponding vertices v_i and v_k is oriented from v_i to v_k. ♦

The oriented *RKPian graph* is shown in Figure 2.

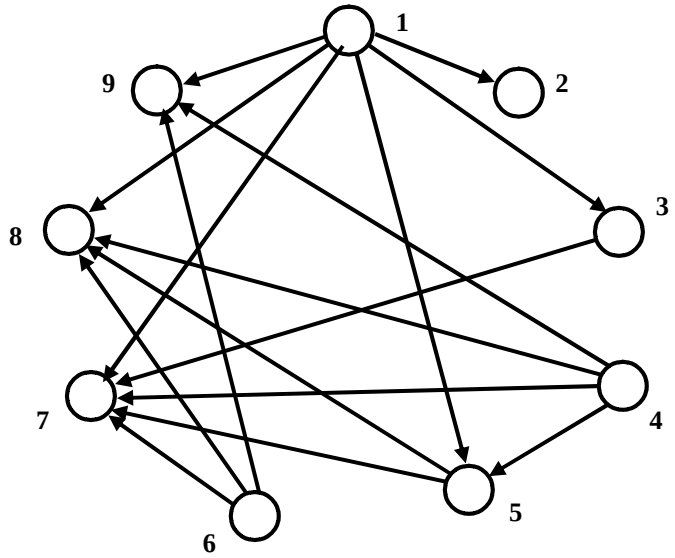

Figure 2. The oriented *RKPian graph* for the given sequence $\Pi = \langle 6\ 4\ 1\ 9\ 5\ 8\ 3\ 7\ 2 \rangle$. An edge $\{v_i,v_j\}$ is oriented from v_i to v_j (*i.e.*, $v_i \rightarrow v_j$), if $\Pi_i \leq \Pi_j$ and $i < j$ in Π.

In fact, there are at least two such orientations in an *RKPian graph*. The notion of the transitive orientability of such a graph is straightway obtained following the proof of Lemma 1, above. In our sorting algorithm, we introduce the natural transitive orientation [6], as directed in this lemma, on the edge joining the corresponding vertices of Π_i and Π_j, from Π_i to Π_j, if $i < j$

and $\Pi_i \leq \Pi_j$, in computing a sorted sequence in non-decreasing order. The interesting transitive property of the oriented *RKPian graph* concludes the following theorem.

Theorem 1. *RKPian graph is a comparability graph.*

Proof. According to the construction of the *RKPian graph*, if i is less than j and Π_i is also less than or equal to Π_j, then only there is an edge between the corresponding vertices v_i and v_j of the graph. So, for any three elements Π_i, Π_j, and Π_k in Π, if $\Pi_i \leq \Pi_j$ and $\Pi_j \leq \Pi_k$ for their respective positions i, j, and k, such that $i < j$ and $j < k$, edges between the corresponding vertices of Π_i and Π_j, and Π_j and Π_k signify that there is an edge between the corresponding vertices of Π_i and Π_k in the graph. ♦

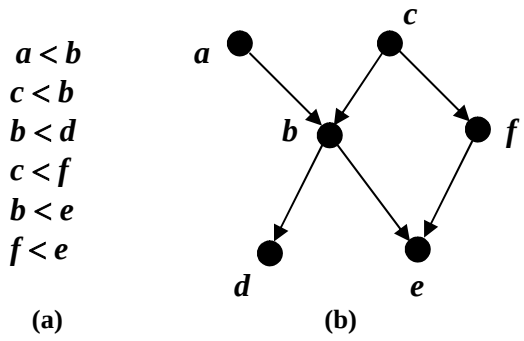

Figure 3. (a) Six partial orders between the keys *a* through *e*. **(b)** The graph theoretic representation of the partial orders, in the form of directed edges in a graph. This graph is a directed acyclic graph (DAG); otherwise, no sorted sequence is computable. According to the topological sorting algorithm, we may have several sorted sequences; some of which are as the following: (i) *a c f b d e*, (ii) *c f a b e d*, (iii) *a c b f e d*, (iv) *a c b d f e*, (v) *c a f b e d*, etc.

So, the computed *RKPian graph*, a comparability graph, is transitively orientable, and after having the natural transitive orientation [6] (as stated above), we obtain an *oriented RKPian graph* with three sorts of vertices: a set of *source* vertices, a set of *intermediate* vertices, and a set

of *sink* vertices. As for example, in Figure 2, vertices 1, 4, and 6 are source vertices, vertices 3 and 5 are intermediate vertices, and the remaining vertices 2, 7, 8, and 9 are sink vertices.

Now we apply the topological sorting algorithm [3, 5] to compute the desired sorted sequence in non-decreasing order, as stated below. In topological sorting, we consider the partial orders $v_i < v_j$, often denoted by directed edges (v_i,v_j) of an oriented graph, and assign v_i earlier than v_j, in computing a sorted sequence. This has clearly been explained in Figure 3.

The topological sorting algorithm is an iterative algorithm. In each iteration, it selects an element to include it and update the sorted sequence. So, for a set of partial orders of n elements, $\Theta(n)$ iterations are required.

In computing a topologically sorted sequence, we first consider an element corresponding to a source vertex in the graph. Since a DAG does not contain any cycle, this could be any vertex among the source vertices in the graph. We place the element in the first position of the computed sorted sequence, and delete its corresponding vertex and adjacent edges, if any, from the graph. In the next iteration, we consider an element corresponding to another source vertex in the modified graph, and place it to the second position of the computed sorted sequence. Accordingly, the graph is modified by deleting the vertex and its adjacent edges, if any, for the next iteration, and so on.

Clearly, topologically computed sorted sequence is not unique based on the set of given partial orders, as at the beginning of an iteration, we may have two or more source vertices in the graph. Consequently, among these source vertices, any one (vertex) could be considered for its placement in computing a sorted sequence, in that iteration. As a result, the topological sorting algorithm is not deterministic. So, this algorithm is not sufficient alone and we need to

incorporate further information available in the given sequence, in order to compute the desired sorted sequence, as explained below.

Needless to mention that the relative arrangement of the numbers (or the elements) in the given sequence is somehow captured in the form of directed edges in the oriented *RKPian graph*, as shown in Figure 2. Nevertheless, this graphical representation of the given sequence of numbers lacks a lot in computing a sorted sequence, as it has been exemplified below.

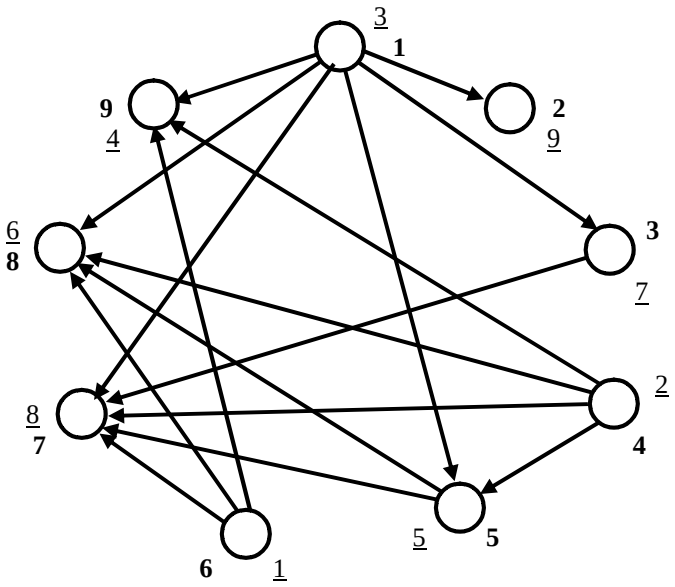

Figure 4. The oriented *RKPian graph* for the given sequence $\Pi = \langle 6\ 4\ 1\ 9\ 5\ 8\ 3\ 7\ 2 \rangle$, associated with the positions (underlined) of the elements in Π.

Here, in Figure 2, in the first iteration, the topological sorting algorithm could select 6 or 4 instead of 1, which is wrong. So, in order to make the algorithm deterministic and compute the desired sorted sequence in non-decreasing order, we introduce the positional values for Π_i in Π (i.e., i), to the corresponding vertices v_i in the oriented *RKPian graph*. Such a graph is shown in Figure 4, where the vertices associate numbers, that are underlined, reflecting positions of the corresponding elements (or numbers) in the given sequence, from left to right.

Here we have an important observation, as stated in the following lemma.

Lemma 2. *The elements in Π corresponding to the source (or sink) vertices in the oriented RKPian graph are sorted in decreasing order.*

Proof. If $\Pi_i \leq \Pi_j$ for $i < j$, then only there is an edge between the vertices v_i and v_j, corresponding to Π_i and Π_j, respectively. Moreover, v_i is an ancestor of v_j, according to the natural orientation on the edges of the *RKPian graph*. In other words, all the source (or sink) vertices in the graph are out of order. So, they are arranged in decreasing order. ♦

Consequently, we conclude the following lemma, at the beginning of the first iteration. Even this is true in general, at the beginning of each of the subsequent iterations when the modified oriented *RKPian graph* is considered for identifying the desired element and updating the sorted sequence.

Lemma 3. *The positional value of the smallest element in Π is maximum amongst the positional values of all the elements whose corresponding vertices are source vertices in the oriented RKPian graph.*

Proof. Following the proof of Lemma 2, all the elements in Π corresponding to the source vertices in the oriented *RKPian graph* are arranged in decreasing order. Therefore, the source vertex (in the graph) with the highest positional value in the given sequence must be the smallest element in Π. ♦

We can easily verify the fact from the oriented *RKPian graph* in Figure 4. Here the positional value of number 6 is 1, number 4 is 2, and that of number 1 is 3 in Π.

So, in the first iteration, we compute the smallest element, which belongs to the maximum position in Π, amongst the source vertices of the oriented *RKPian graph*. Then we modify the graph by deleting the vertex corresponding to this smallest element and the adjacent edge(s), if any, of this vertex from the graph. The modified oriented *RKPian graph* is obtained for computing the second smallest element in Π, in the next iteration. This modified oriented *RKPian graph* for the given sequence $\Pi = \langle 6\ 4\ 1\ 9\ 5\ 8\ 3\ 7\ 2 \rangle$ is shown in Figure 5.

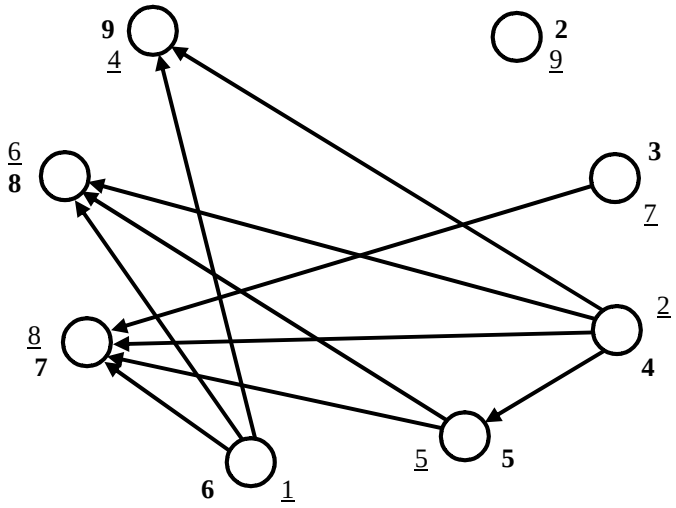

Figure 5. The oriented *RKPian graph* at the beginning of the second iteration of algorithm *RKPianGraphSort* for the given sequence $\Pi = \langle 6\ 4\ 1\ 9\ 5\ 8\ 3\ 7\ 2 \rangle$, associated with the positions (underlined) of the numbers in Π.

Clearly, we have four source vertices, 6, 4, 3, and 2, in this graph, and vertex 2 is at the highest position in Π, i.e., 9, amongst the positional values of all the source vertices in the graph. So, 2 is selected and appended after 1 to update the computed sorted sequence in non-decreasing order. In this way, the ith smallest element in Π is computed amongst the source vertices of the modified oriented *RKPian graph* that is obtained at the beginning of the ith iteration, and so on.

Needless to mention that the initially computed *RKPian graph* is an undirected graph, and the natural transitive orientation on the edges of this graph results a directed acyclic graph (DAG). So, the algorithm, *RKPianGraphSort* always terminates, exactly after n iterations, outputting a sorted sequence of the elements in Π in non-decreasing order. In the next section, we study the stability of the algorithm, and compute the computational complexities of the same.

3. Computational Complexity of the Sorting Algorithm

In this section, we first study the stability of the algorithm. A sorting algorithm is *stable* if elements with equal keys are left in the same order as they occur in the input sequence [1, 3, 5, 7]. To verify the stability of the sorting algorithm developed in this paper, for instance we consider the following example sequence.

4 2 1 4 3 5 4 3 1

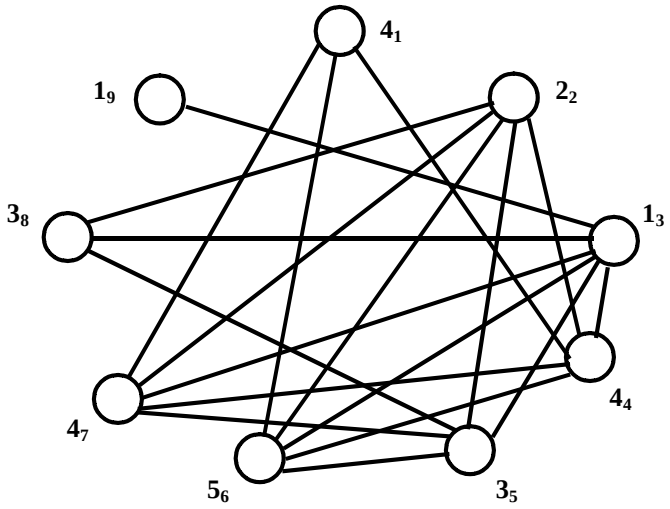

Figure 6. The *RKPian graph* for the example sequence $\Pi = \langle 4\ 2\ 1\ 4\ 3\ 5\ 4\ 3\ 1 \rangle$.

To differentiate the same numbers, let us include a suffix to each of the numbers indicating the position of a number in the given sequence, as follows.

4_1 2_2 1_3 4_4 3_5 5_6 4_7 3_8 1_9

Now according to algorithm *RKPianGraphSort*, we have the *RKPian graph* as shown in Figure 6. Then the graph is oriented as stated in the algorithm. The resulting oriented *RKPian graph* is obtained as shown in Figure 7.

Initially, in Figure 7, we have three source vertices 4_1, 2_2, and 1_3, out of which 1 is at the highest position in Π, i.e., 3. So, we include this 1 and start forming the sorted sequence. Accordingly, modify the oriented graph, and again obtain three source vertices 4_1, 2_2, and 1_9, out of which 1 is at the highest position in Π, i.e., 9. So, we include this 1 and update the sorted sequence. The interesting phenomenon to be noticed is that the relative positions of two 1's, i.e., 1_3 and 1_9 (at their positions 3 and 9, respectively, in Π), are also maintained in computing the sorted sequence. Needless to mention that, eventually, the desired sorted sequence computed by algorithm *RKPianGraphSort*, suffixed with their positions in Π, is as follows.

1_3 1_9 2_2 3_5 3_8 4_1 4_4 4_7 5_6

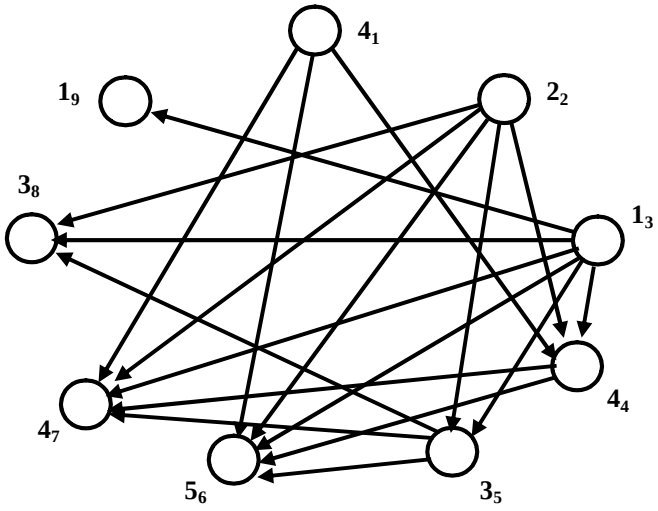

Figure 7. The oriented *RKPian graph* for the example sequence $\Pi = \langle 4\ 2\ 1\ 4\ 3\ 5\ 4\ 3\ 1 \rangle$, suffixed with the positions of the numbers in Π.

Hence we conclude the following lemma.

Lemma 4. *Algorithm RKPianGraphSort is a stable sorting algorithm.*

Proof. In computing the *RKPian graph*, we introduce an edge between the vertices v_i and v_j, only when the corresponding elements $\Pi_i \leq \Pi_j$ for $i < j$. This means that two distinct elements of the same value in Π do not belong to the set of source vertices in an iteration; rather in the oriented *RKPian graph*, v_i becomes an ancestor of v_j. So, v_i belongs to a set of source vertices, and selected for the sorted sequence, earlier than v_j. ♦

Now we study the computational complexities of algorithm *RKPianGraphSort*, developed in this paper. The worst-case complexity of the algorithm is stated in the following theorem.

Theorem 2. *Algorithm RKPianGraphSort runs in time* $\Theta(n^2)$ *in the worst-case, where n is the number of records in the given list to be sorted.*

Proof. The algorithm consists of two parts: an initial part of computation and an iterative part. In the initial part, the *RKPian graph* and its oriented counterpart are computed for the given sequence Π, and the positions of elements Π_i in Π, $1 \leq i \leq n$, are determined. A line sweep algorithm computes the graphs as the following. For n elements in Π, we introduce n isolated vertices into each of the graphs. To introduce edges in the graphs, we consider the elements Π_i in Π, $1 \leq i \leq n-1$, and add edges between v_i and v_j, where Π_j is on the right of Π_i, i.e., $i < j$, such that $\Pi_i \leq \Pi_j$. Accordingly, the edges are oriented from v_i to v_j in computing the oriented *RKPian graph*. Then we label vertices v_i of this oriented graph with Π_is' positions in Π. All these computations of the initial part of the algorithm take time $\Theta(n^2)$, where n is the size of Π.

Now we compute the complexity of the iterative part of the algorithm as follows. At the beginning of each iteration, the source vertices in the oriented (updated) *RKPian graph* and the vertex with the highest position in Π among the source vertices are computed in time $O(n)$. Modification of this oriented graph by deleting the selected vertex and associated edges, if any, to prepare it for the next iteration, could also be done in $O(n)$ time. As the algorithm iterates $\Theta(n)$ times, therefore, the overall worst-case running time of algorithm *RKPianGraphSort* is $\Theta(n^2)$. ♦

The worst-case complexity of the algorithm does not depend on a given sequence of elements (to be sorted) in Π. This is because that the initial computation of the *RKPian graph* and its oriented counterpart take time $\Theta(n^2)$ for a sequence Π of size n. Hence in general, the initial computation of the graphs dominates the overall computational complexity of the algorithm. The best-case running time of the iterative part of algorithm *RKPianGraphSort* is $\Theta(n)$, when a list of n elements is given in sorted (i.e., in non-decreasing) order. In this case in an iteration we have only one source vertex (in the oriented *RKPian graph* obtained in that iteration) to be considered in computing a sorted sequence in non-decreasing order.

4. Conclusion

In this paper, we have developed a new, graph based comparison sorting algorithm *RKPianGraphSort*, that sorts a given list in non-decreasing order and takes time $\Theta(n^2)$ in the worst-case, where n is the number of records to be sorted in the given list. The *RKPianGraphSort* is a stable sorting algorithm.

References

[1] Aho A. V., J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley: An Imprint of Addison Wesley Longman, Inc., Reading Massachusetts, 1999.

[2] Brassard G. and P. Bratley, *Fundamentals of Algorithmics*, Prentice-Hall of India Pvt. Ltd., New Delhi, 1999.

[3] Cormen T. H., C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, Prentice-Hall of India Pvt. Ltd., New Delhi, 2001.

[4] Golumbic M. C., *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[5] Knuth D. E., *The Art of Computer Programming: Sorting and Searching*, Vol. 3, Second Edition, Addison-Wesley: An Imprint of Pearson Education Asia, New Delhi, 2000.

[6] Pal R. K., *Multi-Layer Channel Routing: Complexity and Algorithms*, Narosa Publishing House, New Delhi (Also from CRC Press, Boca Raton, USA and Alpha Science International Ltd., UK), 2000.

[7] Weiss M. A., *Data Structures and Algorithm Analysis in C*, Second Edition, Pearson Education Asia, New Delhi, 2002.

Dr. Rajat K. Pal received his B.E. degree in Electrical Engineering in 1985 from Bengal Engineering College, Shibpur, University of Calcutta, and his M.Tech. degree in Computer Science and Engineering in 1988 from the University of Calcutta. From 1988 to 1989 he served as an Assistant Engineer in the Control and Operational Wing of the West Bengal Power Development Corporation Ltd. (WBPDCL) at Kolaghat Thermal Power Project (KTPP) unit. In 1989 he joined as a Research Scholar (SRF and afterwards RA of CSIR) in the Department of Computer Science and Engineering, Indian Institute of Technology (I.I.T.), Kharagpur, and awarded Ph.D. degree in 1996. Since 1994, he is serving the University of Calcutta as a faculty of the Department of Computer Science and Engineering. He worked as the **Head of the Department of Computer Science and Engineering** of the University of Calcutta for two years (2005-2007). Presently, he is at the post of ***Reader*** of the same department.

His major research interests include VLSI design, Graph theory and its applications, Perfect graphs, Logic synthesis, Design and analysis of algorithms, Computational geometry, Parallel computation and algorithms. Dr. Pal has published around 55 technical research papers, and authored a book entitled "**Multi-Layer Channel Routing: Complexity and Algorithms"** that has jointly been published from ***NAROSA Publishing House***, New Delhi, ***CRC Press***, Boca Raton, USA and ***Alpha Science International Ltd***, UK, in September 2000.