AMO - Advanced Modeling and Optimization, Volume 15, Number 1, 2013

CompleteGraphSort is a Complete Graph Structure based New Sorting Algorithm

Rajat Kumar Pal
Department of Computer Science and Engineering
University of Calcutta
92, Acharya Prafulla Chandra Road
Kolkata – 700 009, West Bengal, India
E-mail: pal.rajatk@gmail.com

Abstract

Sorting is a well-known problem frequently used in many aspects in the world of computational applications. Sorting means arranging a set of records (or a list of keys) in some (non-increasing or non-decreasing) order. In this paper, a complete graph structure based comparison sorting algorithm, *CompleteGraphSort* has been proposed 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.

2010 Mathematics Subject Classification: 68R10

1. Introduction: The Sorting Problem

Sorting is a well-known problem in literature [1-3, 5, 7, 9]; the problem is stated as follows:

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

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 also permutes the satellite data as well.

As for example, let us assume that a given unsorted list of keys (or sequence of numbers) is as follows.

25 17 57 5 37 78 29 9 13

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

5 9 13 17 25 29 37 57 78

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 form.

In this paper, we develop a complete graph structure based sorting algorithm *CompleteGraphSort* that sorts the elements (or keys) in a given list, based on their values and irrespective to their positions in the list (whether one is on the left or on the right of the other between a pair of distinct elements) as satellite data. That means the satellite information that the sorting algorithm captures in its graph theoretic modeling is that whether a pair of elements is distinct and different, and at the same time it is irrelevant whether they are in order or in out of order in a given sequence. The computational (time) complexity of this algorithm is $\Theta(n^2)$ in all the cases, where n is the number of records (or keys) in the given sequence (or list) to be sorted.

The paper is organized as follows. In Section 2, we do a brief survey on graph theory and sorting problem. The new sorting algorithm is proposed in Section 3. Section 4 describes the complexity issues of the algorithm developed herein. Some salient features and comparative benefits are discussed in Section 5, and we conclude the paper with few remarks in Section 6.

2. A Brief General Background

On General Graph Theory and Complete Graph: Graph is a mathematical tool or object consisting of two sets, a set V of vertices and a set E of edges, which is written as $G = (V, E)$, where each element of set E is represented by an unordered pair of distinct elements of set V (for the case of simple symmetric graphs).

A graph in which edges have no orientation, i.e., they are not ordered pairs is called an *undirected graph*. A *directed graph* (or *digraph*) is an ordered pair $D = (V, A)$ with V a set whose elements are called *vertices* (or *nodes*), and A a set of ordered pairs of vertices, called *directed edges* (or *arcs*). That is, a directed graph is a graph with orientations assigned on the edges in making its unordered edges ordered. *Complete graphs* have the feature that each pair of vertices has an edge connecting them. Incidentally, the graph under consideration is a general, simple, symmetric graph. As the graph G is a complete graph, it is somehow special in the domain of general graphs. Undirected graphs are also known as *symmetric graphs* as for any edge $\{u, v\}$ of such a graph, $\{u, v\} = \{v, u\}$; the pair of vertices is unordered. A *simple graph* is an undirected graph that has no self loops and no more than one edge connecting any two distinct vertices of the graph. In some other word, in a simple graph the edges of the graph form a set and no two pairs of edges connect a *distinct* pair of vertices. In a simple graph with n vertices every vertex has a degree that is less than n.

On Sorting Problem: 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 (non-increasing or non-decreasing) order. There are several sorting algorithms in the present day world. Out of which, *insertionsort*, *selectionsort* take $O(n^2)$ time in the worst-case whereas *mergesort* has a better asymptotic running time, $\Theta(n\lg n)$, in the worst-case. *Heapsort* sorts n elements in $O(n\lg n)$ time, whereas the worst-case running time of *quicksort* is $O(n^2)$. The average-case running time of quicksort is $O(n\lg n)$, though, it generally outperforms heapsort in practice [3].

Insertionsort, selectionsort, mergesort, heapsort, and quicksort are all comparison sorts: they determine the sorted order of a given sequence by comparing elements, and it has been proved that heapsort and mergesort are asymptotically optimal comparison sorts [1-3, 5, 7, 9], as each of these

two sorting algorithms is a tree structure based (nonlinear) sorting algorithm. Insertionsort and selectionsort are linear structure based sorting algorithms, whereas quicksort is theoretically a tree structure based sorting algorithm though its practical implementation follows a linear structure based comparison sorting.

Although $\Omega(n\lg n)$ is a lower bound on the worst-case running time of any comparison sort of a sequence of n elements, 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 some constant [3].

On Topological Sorting: In graph theory, a *topological sort* (or *topological ordering*) of a directed acyclic graph (DAG) is a linear ordering of its vertices in which each vertex comes before all vertices to which it has outbound (or outgoing) edges. Every DAG has one or more topological sorts.

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 (which is certainly a DAG), and assign v_i earlier than v_j, in computing a sorted sequence. This has clearly been explained in Figure 1.

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

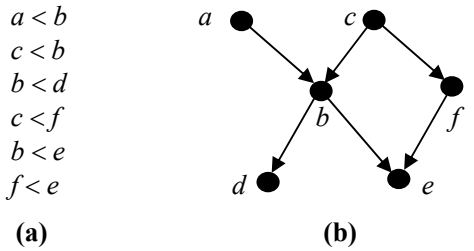

Figure 1: (a) Six partial orders between the keys *a* through *f*. **(b)** The graph theoretic representation of the partial orders, in the form of directed edges in a graph. This graph is a directed acyclic graph (DAG); otherwise, no sorted sequence is computable. According to the topological sorting algorithm we may have several sorted sequences, some of which are as follows: (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.

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 may produce several valid solutions for a given instance of partial orders (see Figure 1). Incidentally, *CompleteGraphSort*, the sorting algorithm proposed in this paper applies topological sorting that computes the desired sorted sequence for a DAG G^* (for the given sequence, or any of its permutations) where we have only one source vertex (and only one sink vertex) at the beginning of

each iteration. This is because the computed graph structure G is always complete for any given sequence Π of length n and G^* is completely transitive from a single source vertex to a single sink vertex through exactly n–2 intermediate vertices, where no two such vertices having the same indegree (or outdegree).

3. The Sorting Algorithm

Let $\Pi = \langle a_1, a_2, \ldots, a_n \rangle$ be the given sequence (or list) of n *unsorted* elements (or keys). The algorithm proposed in this paper sorts the elements in Π in non-decreasing order by computing a complete graph structure as stated below. The algorithm consists of two parts: an initial part of construction of graphs and an iterative part of computing the desired sorted sequence.

For each element a_i in Π, we introduce a vertex v_i to the graph; hence (initially) the graph contains exactly n isolated vertices, where $1 \le i \le n$. Now we introduce edges to the graph, obeying the following logic. For the sake of simplicity we assume that the entire elements in Π are distinct (and different). Later we would consider a case where Π may contain several same keys. So, we compute a complete graph G of n vertices corresponding to the n elements in Π (see Figure 2 for the sequence assumed in Section 1). In other words, we introduce an edge between vertices v_i and v_j, if the corresponding elements a_i and a_j are in (ascending) order by scanning the sequence Π from left to right. In a similar way, we also introduce an edge between vertices v_k and v_l, if the corresponding elements a_k and a_l are in (ascending) order by scanning the sequence Π from right to left. So, in the later case the elements a_k and a_l are in reverse (or descending) order, if we scan Π from left to right. In this way a complete graph is obtained (see Figure 2) as the elements in Π are distinct.

Note that the computed graph, G is transitively orientable [4, 8], as stated below.

Lemma 3.1. *For any three vertices v_i, v_j, and v_k in the computed graph G corresponding to the elements a_i, a_j, and a_k in Π, such that $a_i < a_j$ and $a_j < a_k$ at any positions i, j, and k in Π, 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 G and this edge is oriented from v_i to v_k (i.e., $v_i \rightarrow v_k$).*

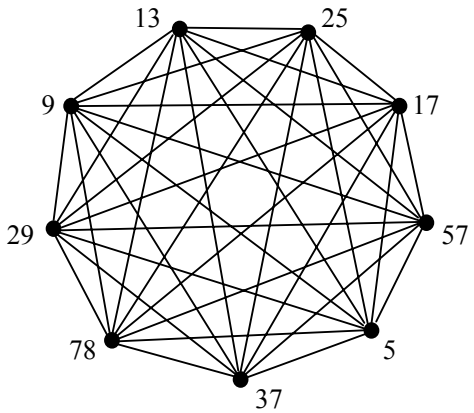

Figure 2: The complete graph G for the given sequence $\Pi = \langle 25\ 17\ 57\ 5\ 37\ 78\ 29\ 9\ 13 \rangle$ to be sorted.

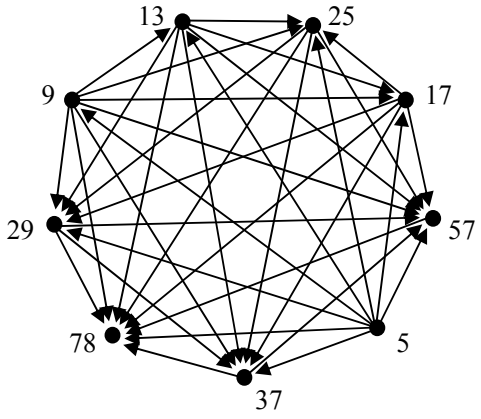

Figure 3: The directed acyclic graph (DAG) G^* obtained by orienting each of the edges of the complete graph G (computed in Figure 2) for the given sequence $\Pi = \langle 25\ 17\ 57\ 5\ 37\ 78\ 29\ 9\ 13 \rangle$ to be sorted.

Now we make the graph directed in the way it is orientable, as stated in Lemma 3.1 above, and obtain a directed acyclic graph (or DAG). We orient an edge $\{v_i, v_j\}$ from v_i to v_j, i.e., we make $\{v_i, v_j\}$ to (v_i, v_j), only if the value of a_i is less than the value of a_j in Π. Needless to mention that we orient an edge $\{v_k, v_l\}$ from v_l to v_k, i.e., we make $\{v_k, v_l\}$ to (v_l, v_k), only if the value of a_k is greater than the value of a_l in Π. Hence we orient the graph, which is absolutely a natural orientation (from a vertex with smaller key value to a vertex with larger key value in Π); the oriented graph G^* is shown in Figure 3. Hence, we conclude the following theorem.

Theorem 3.2. *Since G is a complete graph, so after orienting the edges of G from v_i to v_j, if a_i is less than a_j in Π, the (directed) graph (i.e., G^*) obtained is entirely transitively closed from the source vertex v_s to the sink vertex v_t, where the maximum path length from v_s to v_t is exactly n (in terms of vertices) for a given sequence Π of n elements.*

Proof. The proof is straightforward from Lemma 3.1 above. Pointless to mention that G is a complete graph of n vertices as Π contains n (distinct) elements. We orient an edge $\{v_i, v_j\}$ in G from v_i to v_j, if a_i is less than a_j in Π; otherwise, we orient $\{v_i, v_j\}$ from v_j to v_i. So, if v_s (v_t) is the vertex corresponding to the smallest (largest) element in Π, then for any of the remaining $n-2$ vertices v_r, the corresponding element $a_r > a_s$ but $a_r < a_t$ in Π. Hence we orient $\{v_s, v_r\}$ from v_s to each such vertex v_r and $\{v_r, v_t\}$ from each such vertex v_r to v_t. Moreover, for any three among $n-2$ v_r vertices, say v_{ra}, v_{rb}, and v_{rc}, we orient $\{v_{ra}, v_{rb}\}$ from v_{ra} to v_{rb}, if $a_{ra} < a_{rb}$, orient $\{v_{rb}, v_{rc}\}$ from v_{rb} to v_{rc}, if $a_{rb} < a_{rc}$, and thus we orient $\{v_{ra}, v_{rc}\}$ from v_{ra} to v_{rc}, as $a_{ra} < a_{rc}$. Needless to mention that we also orient the edge $\{v_s, v_t\}$ in G from v_s to v_t, as a_s is less than a_t in Π. Hence we conclude the theorem. ♦

Now it is straightforward (as well as motivating) to point out the significant observations in the oriented graph G^*. These are enlisted in the form of lemmas and corollaries as follows.

Lemma 3.3. *There is only one source (sink) vertex in the oriented graph G^*, which is the smallest (largest) element in Π.*

A *source* (*sink*) *vertex* is a vertex in a directed graph whose indegree (outdegree) is zero. Here *indegree* (of a vertex) is defined as the number of incoming edges to a vertex in G^*. Similarly, outdegree (of a vertex) can also be defined as follows: *Outdegree* of a vertex v_i in G^* is the number of outgoing edges from v_i to some other vertices in the graph.

Lemma 3.3 follows the following corollary.

Corollary 3.4. *Neither of the remaining elements in Π is the smallest or largest in Π, as for each of these elements in Π the corresponding vertex in G^* is an intermediate vertex.*

For an *intermediate vertex* in a directed graph neither the indegree nor the outdegree is zero.

Lemma 3.5. *As G is a complete graph and G^* is a directed acyclic graph (DAG), if the indegree of vertex v_i is p, then a_i would get the $(p+1)$th position if the elements in Π are sorted in ascending order.*

Following Lemma 3.5, we find the beauty (as well as the novelty) of the sorting algorithm *CompleteGraphSort* as follows. If the elements in Π are distinct (and different), as soon as G^* is computed, we can easily identify the kth smallest (or the kth largest) element in Π, $1 \le k \le n$, prior to computing the sorted sequence (in some order). Corollaries 3.6 and 3.7 below tell about the same.

Corollary 3.6. *The indegree (outdegree) of the vertex in G^* corresponding to the kth smallest element in Π is $k-1$ $(n-k)$, $1 \le k \le n$.*

Corollary 3.7. *The indegree (outdegree) of the vertex in G^* corresponding to the kth largest element in Π is $n-k$ $(k-1)$, $1 \le k \le n$.*

This is because separately the indegree and also the outdegree of each of the vertices in G^* is distinct (and different), and they vary from 0 through $n-1$. Moreover, the summation of indegree and outdegree for each of the vertices in G^* is exactly $n-1$.

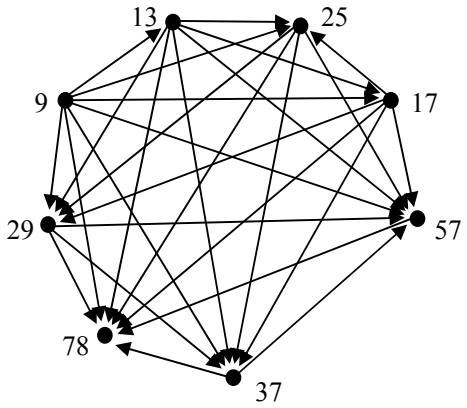

Figure 4: The modified directed acyclic graph (DAG) G^* obtained after deleting 5 as the smallest element in the given sequence $\Pi = \langle 25\ 17\ 57\ 5\ 37\ 78\ 29\ 9\ 13 \rangle$ to be sorted.

However, this completes the initial part of construction of graphs. Then we move into the iterative part of the algorithm starting with G^*. Note that at the beginning of each iteration (initially) in G^* and afterwards, in the modified G^*, we have only one source vertex to be considered. So, the basic topological sorting algorithm suffices to follow over G^* (or over the modified G^*) in order to compute a desired sorted sequence [1-3, 5, 9]. A few steps of the iterative part of the algorithm are shown and explained below.

The iterative part of the algorithm starts with G^* as it is in Figure 3. Here 5 is the (only) source vertex; so we delete it (and its associated edges), and obtain the modified G^* as shown in Figure 4.

Note that 5 is the smallest element in Π. After deleting its corresponding vertex and the associated edges from G^* we obtain the modified G^* as shown in Figure 4 that now contains the only one source vertex 9. So in the next iteration we delete 9 and its associated edges to obtain the modified G^* after the second iteration, as shown in Figure 5. In order to compute a sorted sequence in ascending order, we place 9 after 5 in some array.

This is how the algorithm goes on for n iterations and computes a sorted sequence as we desire. Certainly, 13 is the third smallest, 17 is the fourth smallest, and so on, that are identified at the beginning of next subsequent iterations and deleted along with their adjacent edges till the graph exhausts.

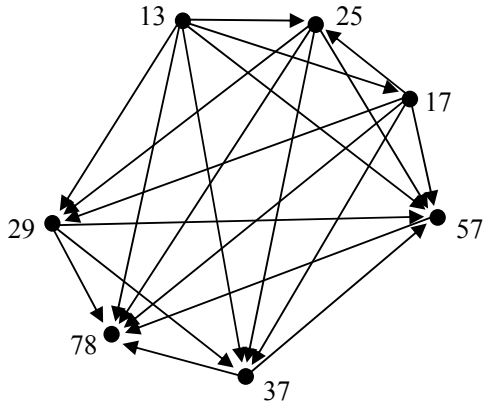

Figure 5: The modified directed acyclic graph (DAG) G^* obtained after deleting 9 as the second smallest element in the given sequence $\Pi = \langle 25\ 17\ 57\ 5\ 37\ 78\ 29\ 9\ 13 \rangle$ to be sorted.

This completes the simplest version of the algorithm as and when Π contains only distinct elements. Complicacy may arise if the elements in Π are not distinct; that means some of them may repeat, for example, as it is assumed below.

24 36 55 5 36 48 13 5 36

Here the desired sorted sequence in ascending order is as follows.

5 5 13 24 36 36 36 48 55

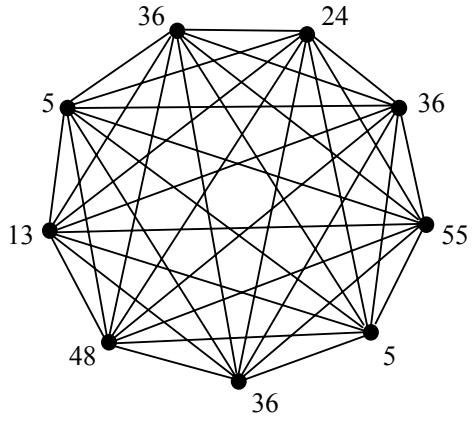

Figure 6: The complete graph G for the given sequence $\Pi = \langle 24\ 36\ 55\ 5\ 36\ 48\ 13\ 5\ 36 \rangle$ to be sorted.

In our proposed sorting algorithm *CompleteGraphSort*, we do the following modifications in computing G and G^* as shown in Figures 6 and 7, respectively. At the same time in making the proposed sorting algorithm stable, we do the following. Note that a sorting algorithm is called a *stable* sorting algorithm, if the elements with equal keys (or values) are left in the same order as they occur in the given input sequence [1, 3, 5, 7, 9]. So, if the elements in the given sequence are differentiated by their positions in Π as suffixed after each element below,

$$24_1 \quad 36_2 \quad 55_3 \quad 5_4 \quad 36_5 \quad 48_6 \quad 13_7 \quad 5_8 \quad 36_9$$

then a stable sorting algorithm would always compute the following sorted sequence in ascending order.

$$5_4 \quad 5_8 \quad 13_7 \quad 24_1 \quad 36_2 \quad 36_5 \quad 36_9 \quad 48_6 \quad 55_3$$

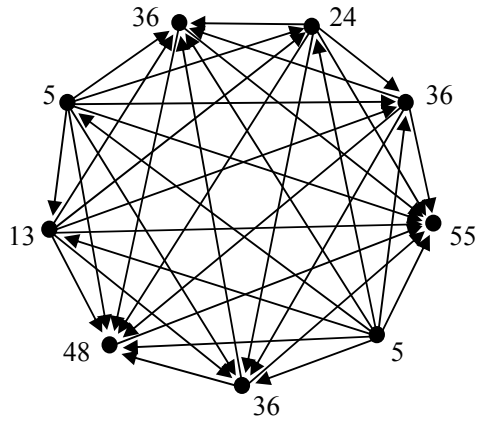

Figure 7: The directed graph G^* obtained from G (computed in Figure 6) for the sequence $\Pi = \langle 24\ 36\ 55\ 5\ 36\ 48\ 13\ 5\ 36 \rangle$ under consideration.

In order to establish our claim that *CompleteGraphSort* is a stable sorting algorithm, we do the following in computing G^*. Certainly, G is a complete graph as shown in Figure 6. Here the elements in Π starting with 24 are introduced clockwise as vertices of G.

In computing G^* from G, what we do, we orient an edge $\{v_i, v_j\}$ from v_i to v_j, i.e., we make $\{v_i, v_j\}$ to (v_i, v_j), only if the value of a_i is less than or equal to the value of a_j (i.e., $a_i \leq a_j$) in Π in scanning the sequence from left to right. Whereas, in the other case, when we consider the sequence from right to left, we orient an edge $\{v_k, v_l\}$ from v_l to v_k, i.e., we make $\{v_k, v_l\}$ to (v_l, v_k), only when the value of a_k is greater than the value of a_l (i.e., $a_k > a_l$) in Π. Hence, we obtain the oriented complete graph G^* for the above unsorted sequence, as it is shown in Figure 7, that does not contain any *two-cycle* [6], i.e., a cycle between a pair of vertices (whose corresponding elements contain the same key value in Π). The remaining part (i.e., the iterative part) of the algorithm is same as before. As a result, we conclude the following lemmas.

Lemma 3.8. *The CompleteGraphSort always computes a complete graph of n vertices at the constructive part of the sorting algorithm for any given sequence Π of n elements, where some of the elements may repeat.*

Lemma 3.9. *The CompleteGraphSort is a stable sorting algorithm.*

After each iteration, the smallest and/or the largest key(s) is (are) identified and deleted from the graph in order to place it (them) in its (their) own position(s) in the desired sorted sequence being computed. The vertex (vertices) and its (their) associated edges are then deleted to start with the next iteration till the graph exhausts.

Needless to mention further that the initially computed graph G is an undirected complete graph and a sort of natural (transitive) orientation on the edges of this graph results a directed acyclic graph (DAG) G^*. So, the sorting algorithm proposed in this paper always terminates, exactly after n iterations, outputting a sorted sequence of the elements in Π in non-decreasing order. The method can also be used for computing a sorted sequence in non-increasing order, when we select and delete the sink vertex (and its associated edges) from the oriented graph in the ith iteration for its position (i.e., the ith position) in the sorted sequence, $1 \leq i \leq n$.

4. An Improved Version of the Sorting Algorithm and Its Complexity Issues

Now we mention an important improvement of the proposed sorting algorithm so that the number of iterations in computing a desired (sorted) sequence is reduced to $\lceil n/2 \rceil$ for a given sequence of n elements. Here in the ith iteration, instead of deleting only the ith smallest element (i.e., the source vertex at the beginning of this iteration) or only the ith largest element (i.e., the sink vertex at the beginning of this iteration), we delete both of them (along with their adjacent edges) for their respective positions in the computed sorted sequence, where $1 \leq i \leq \lfloor n/2 \rfloor$. If n is odd, then in the $\lceil n/2 \rceil$th iteration we place the remaining element to the middlemost position of the sorted sequence. In fact, a bit later we will see that only $\lfloor n/2 \rfloor$ iterations are sufficient in computing the desired sorted sequence of length n. This improvement is eventually obtained from Lemmas 3.3 and 3.5. Following Figure 3, we briefly state it as follows.

Note that 5 (78) is the only source (sink) vertex in G^* (see Figure 3). In our algorithm, in the first iteration we delete 5 (78) (and their associated edges) to place it in the first (last) position in the sorted sequence we like to compute. Hence we obtain the modified oriented graph, as shown in Figure 8, at the beginning of the second iteration. From this figure it is clear that 9 and 57 are the next

vertices for their own positions (that are the second smallest and second largest) in the desired sorted sequence. So, we delete them in the second iteration and in this way after execution of the fourth iteration the only element 25 remains to place it in the middlemost position of the desired sorted sequence being computed; the middlemost position being the $\lceil n/2 \rceil$th position for a given sequence of n elements (where n is odd). Here the middlemost position is the fifth position, as the number of iterations already executed is $\lfloor n/2 \rfloor = \lfloor 9/2 \rfloor = 4$ for the given sequence Π of 9 elements only. Hence only $\lfloor n/2 \rfloor$ iterations are sufficient to compute a desired sorted sequence of n elements. This is because after the execution of the $\lfloor n/2 \rfloor$th iteration, only one element is there to place it in the middlemost position of the sorted sequence (if n is odd). We conclude the result in the following lemma.

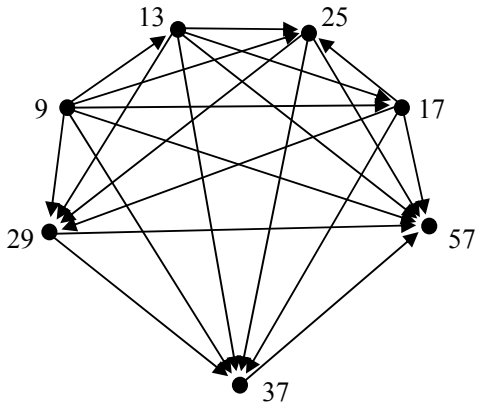

Figure 8: The modified directed acyclic graph (DAG) G^* obtained after deleting 5 and 78 as the smallest and the largest elements in the given sequence $\Pi = \langle 25\ 17\ 57\ 5\ 37\ 78\ 29\ 9\ 13 \rangle$ to be sorted.

Lemma 4.1. *The sorting algorithm CompleteGraphSort requires $\lfloor n/2 \rfloor$ iterations in the worst-case in computing a sorted sequence for any given sequence Π of n elements.*

Even for the improved version of the algorithm, where the number of iterations is $\lfloor n/2 \rfloor$, the algorithm remains stable. Now we study the computational complexities of the algorithm developed in this paper. The worst-case complexity of the algorithm is stated in the following theorem.

Theorem 4.2. *The sorting algorithm CompleteGraphSort requires $\Theta(n^2)$ time in the worst-case, where n is the number of records (or keys) in the given list (or sequence) to be sorted.*

Proof. The algorithm consists of two parts: an initial part of computation of the complete graph G and its oriented counterpart G^*, and an iterative part. The initial part of the algorithm takes time $\Theta(n^2)$, where n is the size of the given sequence Π. On the other hand, the worst-case time taken by the iterative part of the algorithm is $\Theta(n)$. Hence, the overall worst-case running time of *CompleteGraphSort* is $\Theta(n^2)$. ♦

This is worth mentioning that the best-case, average-case, and worst-case running time of the iterative part of the algorithm is $\Theta(n)$, when a set of n elements is given in any order in Π.

5. Comparative Benefits and Advantages

Here we briefly state some salient features of the sorting algorithm developed in this paper. The proposed sorting algorithm is a complete graph structure based sorting algorithm. This is the first

complete graph structure based sorting algorithm in the world. Yes, it is also a comparison sorting algorithm for which it takes the worst-case computational time complexity $\Theta(n^2)$ for a given list of n elements. Graph structures that are computed in developing a sorting algorithm designed earlier were at most trees but never complete graphs, except the *RKPianGraphSort* [7].

The *RKPianGraphSort* is a perfect graph based modeling in designing a sorting algorithm that actually captures the inherent satellite information between a pair of elements in a given sequence, which is represented in the form of an edge (either it is there or it is not there) in computing the perfect graph. In practice, the perfect graph that we obtained in the *RKPianGraphSort* is a comparability graph [4, 8]. In this case the graph we compute is either sparse or dense though the complexity is always $\Theta(n^2)$ time for a sequence of n elements, which is asymptotically tight. Incidentally, the iterative part of this algorithm might take time $O(n^2)$ time in the worst-case for a given sequence of length n, whereas the iterative part of *CompleteGraphSort* (the sorting algorithm proposed in this paper) never takes worse than $O(n)$ time. One more thing we like to mention is that the *RKPianGraphSort* uses positional values of the elements (that are associated to the vertices) in the given sequence. Our proposed algorithm *CompleteGraphSort* does not use any positional value of the elements in the given sequence.

Another uniqueness of *CompleteGraphSort* is that it is a sequence independent sorting algorithm; for any permutation of a given set of elements this algorithm computes the same G and the same G^*, though the sorted sequence computed is always a stable sorted sequence following a given permutation of the elements.

The proposed sorting algorithm always takes $\Theta(n^2)$ time in the worst-case for a given sequence of n elements. Moreover, the innovation of the sorting algorithm is that it can easily identify the kth smallest (or the kth largest) element in the given sequence following the (initial) constructive part of the algorithm (i.e., computation of the graph), $1 \leq k \leq n$, prior to computing the sorted sequence (in some order); this task of sorting (from the computed complete graph structure) might take $O(n)$ time in the worst-case. The iterative part of the algorithm is also very fast, which is no worse than $\lfloor n/2 \rfloor$ iterations, in general.

Regarding memory requirement, the proposed complete graph structure based sorting algorithm developed in this paper takes space $O(n^2)$, where n is the number of elements in the given sequence. This representation of the graph is based on n linked linear (adjacency) lists, lead by n distinct vertices of the graph corresponding to n elements in the given sequence. An adjacency matrix representation is also equally good as the graph structure is complete. Hence the space requirement and its manipulation in representing the graph(s) is $O(n^2)$.

As already mentioned that the proposed algorithm is also a stable sorting algorithm like *RKPianGraphSort* [7]. Furthermore, the algorithm requires at most $\lfloor n/2 \rfloor$ iterations to sort a given sequence of n elements whereas *RKPianGraphSort* requires exactly n iterations.

6. Conclusion

In this paper, we have developed a new, complete graph structure based comparison sorting algorithm *CompleteGraphSort* that sorts a given list (or sequence) in some sorted order and takes time $\Theta(n^2)$ in the worst-case, where n is the number of records (or keys) to be sorted in the given list. The sorting algorithm developed in this paper is a stable sorting algorithm that requires $\lfloor n/2 \rfloor$ iterations in computing the desired sorted sequence. In addition, the sorting algorithm can compute the kth

smallest or the kth largest element in the given sequence, $1 \leq k \leq n$, prior to computing the sorted sequence in some order.

REFERENCES

1. A.V. Aho, 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. G. Brassard and P. Bratley, **Fundamentals of Algorithmics**, Prentice-Hall of India Pvt. Ltd., New Delhi, 1999.
3. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, **Introduction to Algorithms**, Prentice-Hall of India Pvt. Ltd., New Delhi, 2001.
4. M.C. Golumbic, **Algorithmic Graph Theory and Perfect Graphs**, Academic Press, New York, 1980.
5. D.E. Knuth, **The Art of Computer Programming: Sorting and Searching**, Vol. 3, Second Edition, Addison-Wesley: An Imprint of Pearson Education Asia, New Delhi, 2000.
6. R.K. Pal, **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. R.K. Pal, *RKPianGraphSort: A Graph based Sorting Algorithm*, International Journal of ACM Ubiquity, **8.41** (Oct. 16-22, 2007), 16 pages.
8. J.L. Ramirez Alfonsin and B.A. Reed (Editors), **Perfect Graphs**, John Wiley and Sons Ltd., Chichester, 2001.
9. M.A. Weiss, **Data Structures and Algorithm Analysis in C**, Second Edition, Pearson Education Asia, New Delhi, 2002.