CS 297 Report

Algorithms to obtain a total order from partial orders for social networks

Chandrika Satyavolu

satyavolu.chandrika@gmail.com

Advisor: Dr. Chris Pollett

San José State University
Department of Computer Science
One Washington Square
San Jose, CA 95192-0249
Ph: (408) 924-5145
http://www.cs.sjsu.edu/faculty/pollett/

17th May, 2007

TABLE OF CONTENTS

1. Introduction

Social networks use sorting algorithms to display results of a search in a certain order. This order is based on the ranking that these objects are given by the users of the social networking website. The total order obtained in this way is not very accurate as the objects are ordered on the basis of absolute ranks. The goal of my project is to develop an algorithm that would give a more accurate total order from a set of partial orders as opposed to absolute ranks. These partial orders are acquired from the users of the social network. It is not possible to derive these partial orders from a computer program. Such an example of harnessing human intelligence to solve problems that are hard to program a computer to do is Recaptcha that uses human intelligence to digitize books and enable search through scanned text.

This report describes the background work conducted for the actual implementation of the project during CS298. This background work and the experiment results were arrived at in a course of four deliverables. These deliverables were aimed at acquiring the results that would form the basis of the main project during CS298. The first and second deliverables are algorithms programmed using Java. The first deliverable was to program a sorting network. The second deliverable was to program the partial order sorting algorithm. The partial order sorting algorithm works similar to a quicksort, the basic difference being the partitioning of the total element set that makes use of the partial orders to partition. The third deliverable gives the results of the fault tolerance of the partial order sorting algorithm when a small error is introduced into a certain percentage of partial orders. The fourth deliverable gives the results of the experiments conducted to minimize the product of the number of partial orders and the partial order set size which is essentially the space complexity of the algorithm. This report ends with some ideas for future work that would be taken up and implemented in the main project during CS298.

2. Deliverable 1

The first deliverable was to program a sorting network. A sorting network consists of comparators and wires or inputs going into those comparators. Each comparator takes two inputs and sorts them as shown in Fig 1.

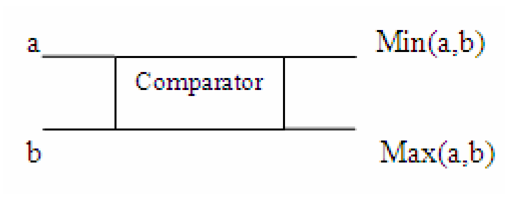

Fig 1.

There are many such comparators operating at the same time in parallel to form one level of a sorting network.

A sorting network makes recursive calls to three subroutines to sort a set of elements. They are:

- Sorter [n]

  It recursively calls the Merger subroutine on n elements.
- Merger[n]

  It recursively calls the Sorter subroutine on two sets of n/2 elements until the sets are of size 2. When the merger is called on sets of size two, it calls the Bitonic-Sorter to merge the two element sets after which it goes to merge four element sets and calls Bitonic-Sorter recursively until it merges the n elements together.
- Bitonic-Sorter[n]

  It recursively calls itself and sorts n elements by comparing ith element with (n/2 +i)th element.

The recursion diagram given in Fig 2. shows the recursive operations of the three subroutines.

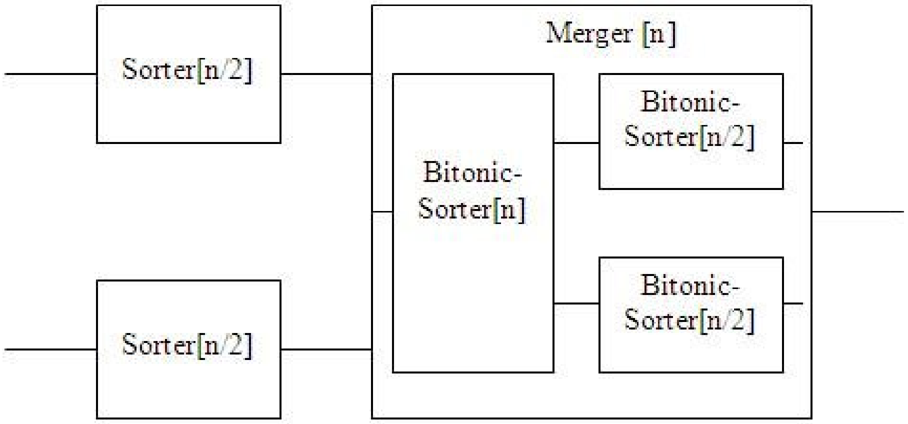

Fig 2.

An example of a sorting network is given in Fig 3. The input list to be sorted (6, 4, 7, 5, 3, 9, 5, 1) is displayed in red on the left end of the network in the Fig 3. The red boxes indicate the Merger[n] subroutine. The comparators inside the red boxes show the Bitonic-Sorter operations. The lines joined with dots form the comparators. The comparators on the same layer are run simultaneously. The intermediate results of the comparators on each layer are shown after each layer. The output of the sorting network (1, 3, 4, 5, 5, 6, 7, 9) is displayed in green on the right end of the sorting network.

The first deliverable consists of the program to generate such a network. The inputs and outputs of the program are:

Inputs:	File containing a list of integers to be sorted
Output:	Sorted list displayed on console.

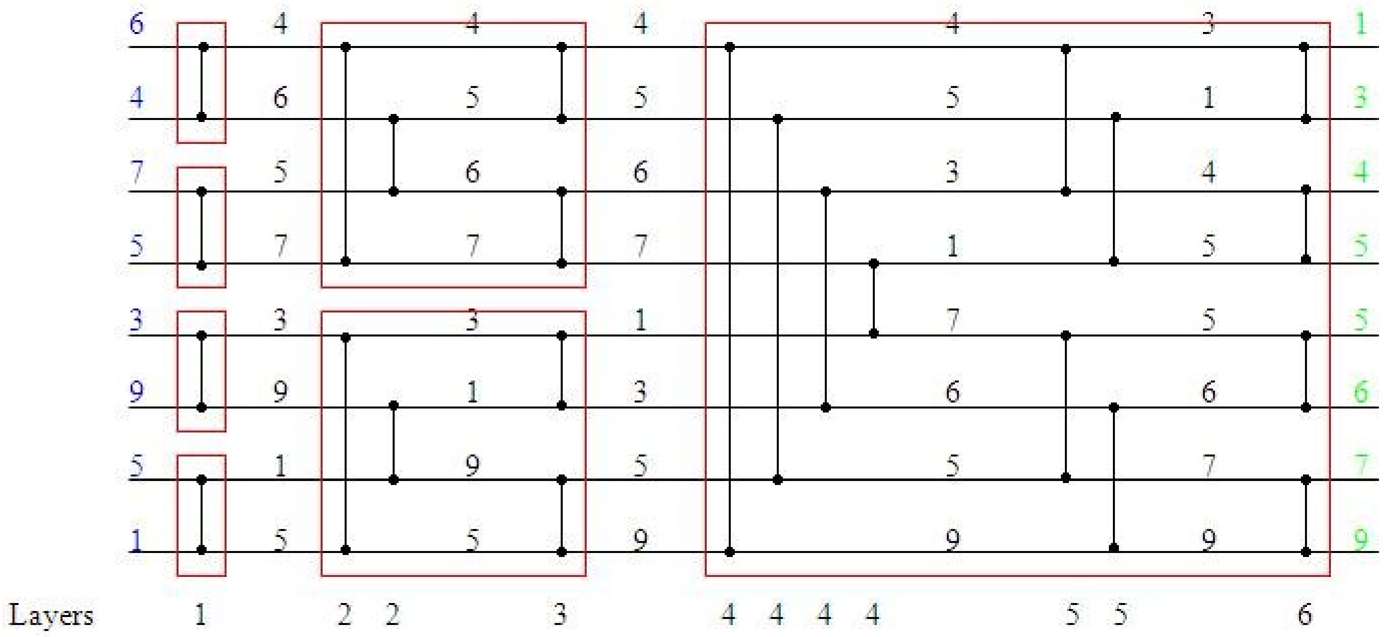

Fig 3.

3. Deliverable 2

The second deliverable was to develop and program a partial order sorting algorithm. The algorithm uses the divide and conquer approach proposed by the planning algorithms where a plan is broken down into smaller plans and built back into a total plan after solving the smaller plans. This partial order sorting algorithm follows a similar approach. The important subroutines in the algorithm are:

- quicksort -
  Recursively sorts the input list based on the sorted partial order sequences.

- partition -
  Partitions the portion of input list into two parts. The last element of the list is chosen as the pivot element. The list is partitioned such that first part has elements that have occurred before the pivot element in most of the partial orders and the second part has numbers that have occurred after the pivot element in most of the partial orders.

- getRank
  Gets the rank of an element in the input list with respect to the pivot element based on partial orders. If it occurs before pivot element more number of times than after in the partial orders, it returns a rank 1 otherwise it returns a rank 0.

The working of the partial order sorting algorithm can be understood best with the help of an example. Fig 4 shows the recursion tree for an input list of elements to be sorted.

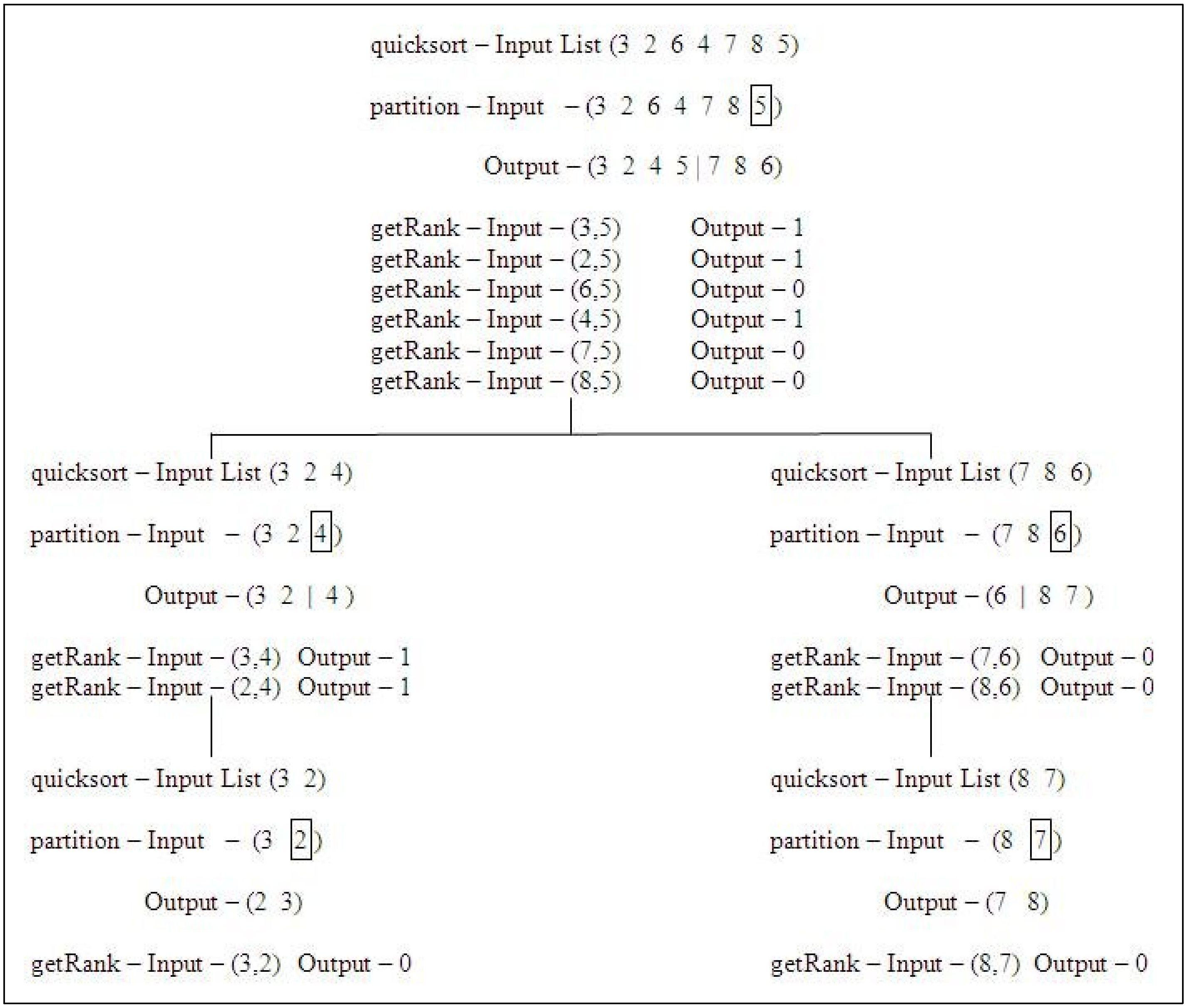
quicksort – Input List (3 2 6 4 7 8 5)
partition – Input – (3 2 6 4 7 8 5)
Output – (3 2 4 5 | 7 8 6)
getRank – Input – (3,5) Output – 1
getRank – Input – (2,5) Output – 1
getRank – Input – (6,5) Output – 0
getRank – Input – (4,5) Output – 1
getRank – Input – (7,5) Output – 0
getRank – Input – (8,5) Output – 0
quicksort – Input List (3 2 4)
partition – Input – (3 2 4)
Output – (3 2 | 4)
getRank – Input – (3,4) Output – 1
getRank – Input – (2,4) Output – 1
quicksort – Input List (3 2)
partition – Input – (3 2)
Output – (2 3)
getRank – Input – (3,2) Output – 0
quicksort – Input List (7 8 6)
partition – Input – (7 8 6)
Output – (6 | 8 7)
getRank – Input – (7,6) Output – 0
getRank – Input – (8,6) Output – 0
quicksort – Input List (8 7)
partition – Input – (8 7)
Output – (7 8)
getRank – Input – (8,7) Output – 0

Fig 4.

Input List:	{3, 2, 6, 4, 7, 8, 5}
Partial Orders:	{{2, 3, 4, 5}, {2, 3, 4, 6}, {3, 4, 6, 7}, {5, 6, 7, 8}, {2, 5, 7, 8}, {3, 6, 7, 8}, {4, 6, 7, 8}}

The recursive subroutine quicksort calls the partition on the input list. The partition subroutine sets the last element of the input list as the pivot element and calls the getRank subroutine on every element (other than pivot) to compare it to the pivot with respect to the relative positions of the comparison element ant the pivot element in the pre-created partial orders. It partitions the input list based on the rank returned by the getRank subroutine. The getRank subroutine checks the partial orders to see how many times the element occurred before the pivot element as opposed to the number of times it occurred after. If it occurs before pivot element more number of times than after in the partial orders, it returns a rank 1 otherwise it returns a rank 0.

The deliverable 2 sorts a list of elements in accordance with the algorithm explained above.

4. Deliverable 3

The third deliverable was to conduct experiments to affirm the fault tolerance of the partial order sorting algorithm. It includes two experiments:

Experiment 1: It focuses on deriving the m (partial order set size), that would give back the correct total order for the case when k = n (total number of elements to be ordered = number of partial orders).

For different values of n (300 >= n >= 20), the number of partial orders is fixed at n (k = n) and the partial order size is fixed at the following percentages of n gives back the total order:

(a) m = 50% of n	20 <= m <= 100
(b) m = 25% of n	100 < m <= 200
(c) m = 20% of n	200 < m <= 300

Observations:

1. The value of m as derived from Experiment 4 in Algorithm Analysis is about the same as the values of m as derived from percentages listed above to arrive at the borderline m that would get just back the total order.

2. Introducing error at these borderline cases of m would give a more accurate outlook on the fault tolerance of the algorithm.

Experiment 2: It introduces a small error into the partial orders for n, k and m values (total number of elements, number of partial orders and partial order set size) fixed at the values derived in experiment 1.

For different values of n (300 >= n >= 20), errors are introduced into partial orders. The error is introduced in a partial order by swapping two elements in a partial order. Here:

p = % of error introduced.
i.e p % of partial orders have single error in them.

The following table lists the number of total orders constructed that are incorrect when a p% error is introduced into the partial orders. It also shows the percentage chance of arriving at an incorrect total order.

	Number of 'n' values giving errors (Total no. of values = 30)	**% error in 30 values of n**
p = 5% error	2	0.6
p = 10% error	3	0.9
p = 20% error	4	1.2
p = 50% error	4	1.2
p = 75% error	10	3
p = 90% error	10	3
p = 100% error	7	2.1

Table 1.

Observations:

1. A small percentage of error (p <= 10%) introduced cannot deduce the fault tolerance property of the algorithm.

2. As the error introduced is a two-element error, there is a good possibility of getting back the correct total order even if the percentage of error is large.

3. Even a 100% two-element error introduced had no greater than 5% chance of the total order being incorrect.

5. Deliverable 4

The fourth deliverable lists the experiments conducted to analyze the space complexity of the algorithm focusing on optimizing the k (no. of partial orders) and m (partial order set size) values for fixed values of n (total number of elements to be ordered). It includes four experiments:

Experiment 1: It derives a relation between k and n (number of partial orders and partial order set size). It is conducted for two cases:

(a) k <= n, m <= n
(b) k – unbounded, m <= n

Observations:

1. The common observation in both cases is that k is inversely proportional to m. Smaller is the partial order size (m), greater is the number of people (k) required to order them to get back the correct total order.

2. The product of k and m (k x m) decides the space required to store the partial orders. This number must be minimized to optimize the space complexity. From the above two tables, we can see that the product k x m is minimum when m is closest to n. However, it is not practical to have partial orders (m) the size of the total number of elements (n).

Experiment 2: It aims at arriving at the function relating k to n for a fixed m. The function is fixed at a constant and the experiment is conducted to see if the total order can be obtained from the partial orders.

For different values of n (300 >= n >= 10), the partial order size is fixed at m = 10 and k is a multiple of n

k = C x n (C is a constant.)

(a) k = n
(b) k = 2 x n
(c) k = 5 x n
(d) k = 10 x n
(e) k = 50 x n

Observations:

1. The first four cases, k = n, k = 2 x n, k = 5 x n and k = 10 x n give back the total orders when the total number of elements (n) is small. However, for larger values, such a k does not get back the total order.

2. The last case where k = 50 x n works for even larger values of n (e.g. n = 290). However, it does not give back the total order for n greater that 300.

3. So, in the function k = C x n, if C is a constant selected from the set of all positive integers, it will give back the correct total order for the cases where C is also derived from a function f(n).

Experiment 3(I): It further examines k to be a function of f(n) x n as established at the end of experiment 2. The constant in experiment 2 is fixed at a function of n (f(n)) and the experiment is conducted to see if the total order can be obtained from the partial orders.
For different values of n (300 >= n >= 10), the partial order size is fixed at m = 10 and k is a multiple of n

k = f(n) x n

(a) f(n) = n	k = n x n
(b) f(n) = n/2	k = n/2 x n
(c) f(n) = sqrt(n)	k = sqrt(n) x n
(d) f(n) = log(n)	k = log(n) x n

Observations:

1. The first two cases k = n x n, k = n x n/2 give back the total order correctly. On the other hand, k = sqrt(n) x n and k = log(n) x n yield very poor results, especially when n gets larger.

2. From these results we can deduce that k must be a function of n such that
   k = f(n) x n
   and
   f(n) = c x n where $0 < c <= 1$

Experiment 3(II): It further examines k to be a function of n^2 as established at the end of experiment 3(I). The function (f(n)) in experiment 3(I) is fixed at a function of n and the experiment is conducted to see if the total order can be obtained from the partial orders.

For different values of n (300 >= n >= 10), the partial order size is fixed at m = 10 and k is a multiple of n

k = C x n^2

(a) C = 1/4	k = 1/4 x n^2
(b) C = 1/6	k = 1/6 x n^2
(c) C = 1/8	k = 1/8 x n^2

Observations:

1. The first case, k = (1/4) n^2, gives back a correct total order every single time. The second case k = (1/6) n^2, give back the total order correctly too most of the time. The third case, (1/8) n^2, does not give back the total order on many occasions.

2. From these results we can deduce that k must be a function of n such that
   k = (1/6) n^2

which also includes $k = (1/4)\ n^2$. The constant C can be safely established to be approximately 0.1667 or 1/6.

Experiment 4: It examines the case where $k = n$ and tries to derive the borderline m that would give back the correct total order from the partial orders.

Observations:

1. The value of m (partial order set size) as a fraction of n (total set size) decreases as the n increases (also k increases with n).

2. The m value is closer to n when n is small (so is k).

6. Future Work

In the CS298 course, the plan is to implement the algorithm in a practical scenario. A high-level design idea would be to implement a web portal that uses this algorithm. An example of such a portal would use the algorithm developed in Deliverable 2 to display the search results. The search results would return the objects closely or remotely related to the search in a specific order. This order is generated from a set of pre-sorted partial orders. The partial orders are obtained after the web portal users sort the partial sets that are randomly generated. The number of partial orders and the partial order set size would be fixed at certain values in accordance with the experiments performed in Deliverables 3 and 4. The total order for the search generated in such a way using the algorithm in Deliverable 2 could also display its partial order rank giving the user the information on the point of view of other users on different objects.

7. Conclusion

The algorithm developed and the research conducted on the figures for number of partial orders and partial order set sizes during CS297 includes most of the groundwork required to implement the final project in CS298. The deliverables were put to test on a small scale and results were derived both theoretically and empirically. The algorithm design took a lot of thinking. There was significant amount of coding involved in all four of the deliverables. The experiments performed kept the algorithm fresh on mind and interesting. The CS297 project has introduced me to several new areas overlapping algorithms and artificial intelligence specifically called the planning algorithms.

8. References

[1] Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Prentice-Hall, 2nd edition, 2002.

[2] Intelligent Planning - A Decomposition and Abstraction Based Approach. Qiang Yang, Springer-Verlag, 1997.

[3] Artifical Intelligence: A Modern Approach. S. Russell and P. Norvig, Prentice-Hall, 2nd edition, 2002.

[4] Depth Optimal Sorting Networks Resistant to k Passive Faults, M Piotrów, Proc. 7th SIAM Symposium on Discrete Algorithms, 1996.

[5] Quicksort example, http://www.cise.ufl.edu/~ddd/cis3020/summer-97/lectures/lec17/sld003.htm