2008

Algorithm to Obtain Total Order from Partial Orders for Social Networks

Chandrika Satyavolu
San Jose State University

ALGORITHM TO OBTAIN TOTAL ORDER FROM PARTIAL ORDERS FOR SOCIAL NETWORKS

A Writing Project
Presented to
The Faculty of Computer Science
San Jose State University
In Partial Fulfillment of the Requirement for the
Degree
Master of Science
By
Chandrika Satyavolu

Dec 2008

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

_____

Dr. Chris Pollett

_____

Dr. Robert Chun

_____

Dr. David Taylor

Abstract

We developed an algorithm for generating total orders from partial orders based on a variant of QuickSort. We also constructed a website: 100 best movies! that would generate and display the total order of the best 100 movies from partial orders that are humanly ordered and stored in a database. The users of the website can add new movies of different categories to the database. The users can also generate partial rankings within specific categories. The total order of 100 best movies! would be calculated from the humanly computed partial orders stored in the database.

Keywords: Social-networks, sorting, absolute-ranks, partial-orders, total-orders.

Table of Contents

1. Introduction……7

    1.1 Project Overview…… 7
    1.2 Report Overview…… 7

2. Background and theory……..8

    2.1 Sorting Network……...8
    2.2 QuickSort Algorithm……... 10
    2.3 Partial Order based QuickSort…… 11
    2.4 Experiments conducted and conclusions drawn…….. 14

        2.4.1 Experiments to affirm fault tolerance……... 14
        2.4.2 Experiments to analyze space complexity of the algorithm…. 16
        2.4.3 Experiments with random data……..……22
        2.4.4 Conclusion of experiments……... 23

3. Software Design……24

    4.1 Tools and Programming Languages ……... 24
    4.2 Implementation and Design……24
    4.2 Database description…… 27
    4.3 Site flow……...30

4. Experiments with users and evolution of the system……40

5. Comparison with existing systems……41

6. Conclusion…… 44

References……45

Index of Figures

Fig 1. Sorting Network Recursion……... 9
Fig 2. Bitonic Sorting Network Example…… 10
Fig 3. Partial Order QuickSort Example…… 13
Fig 4. Plot of k vs. m for n = 100 (k bounded by n)…… 17
Fig 5. Plot of k vs. m for n = 100 (k unbounded)……. 17
Fig 6. Plot for k = n vs. m…….. 21
Fig 7. Software entities and relationships……27
Fig 8. Entity-Relationship diagram……..28
Fig 9. 100 best movies! Homepage……. 30
Fig 10. 100 best movies! SignIn…… 31
Fig 11. 100 best movies! Register……32
Fig 12. 100 best movies! MyProfile…… 33
Fig 13. 100 best movies! Add Movie……...34
Fig 14. 100 best movies! Rank Movies…….. 35
Fig 15. 100 best movies! Create Group……... 36
Fig 16. 100 best movies! Select Group……37
Fig 17. 100 best movies! Select Group……37
Fig 18. 100 best movies! Rank…… 38
Fig 19(a). 100 best movies! Remove Movie……39
Fig 19(b). 100 best movies! Remove Movie……... 40
Fig 20. www.imdb.com…….. 42
Fig 21. www.rankrz.com……..43

Index of Tables

Table 1. Error introduced vs. Fault Tolerance……. 16
Table 2. Software Components……27

Index of Listings

Listing 1. Quicksort Java Code……25
Listing 2. Partition Java Code……. 25
Listing 3. GetRank Java Code……. 26

1 Introduction

1.1 Project Overview

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 a 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. A variant of this algorithm is built into a test website: 100 best movies! This website displays the total order of the best 100 movies from partial orders that are humanly ordered and stored in a database. The users of the website can add new movies of different categories to the database. The users can also generate partial rankings within specific categories. The total order of best 100 movies would be calculated from the humanly computed partial orders stored and obtained from the database.

1.2 Report Overview

This report describes the background work conducted for the actual implementation of the project during CS297 and CS298 project. This background work and the experiment results were achieved by four deliverables as a part of CS297. These deliverables were aimed at acquiring the results that formed 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. The deliverable for CS298 is a test website: 100 best movies! that makes use of the algorithm to get a total order from a set of partial orders. This report ends with some ideas for future work that could be developed and implemented in the test website.

2 Preliminary work

2.1 Sorting Network

The sorting network was studied and implemented for a bitonic sorter during the course of CS297. The sorting network was considered with all the other O(n logn) networks (including QuickSort) for implementing the partial order sorting algorithm. However, we believed that a Sorting Network would be only as good as the QuickSort algorithm in the absence of multiple processors.

A sorting network of a set of comparators connected in parallel and sequentially to sort a set of numbers. It works on the zero-one principle. The zero-one principle states that a sorting network is valid if it can sort all 2^n sequences of 0s and 1s.

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 i_{th} element with $(n/2 + i)_{th}$ element.

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

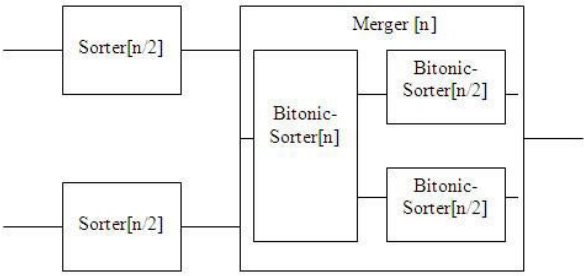

Fig 1. Sorting Network Recursion

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.

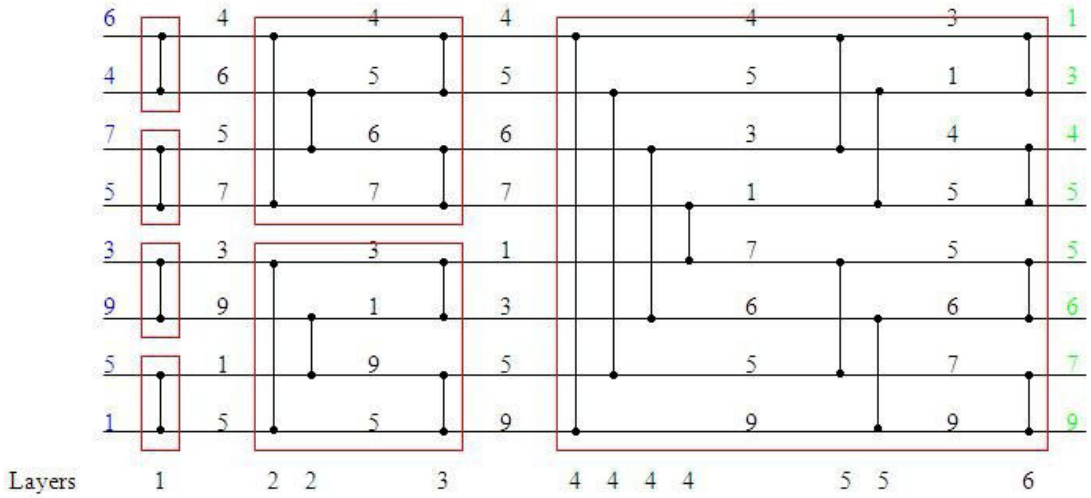

Fig 2. Bitonic Sorting Network Example

2.2 QuickSort Algorithm

QuickSort sorts using a divide and conquer approach to divide a list into two sub-lists. The basic steps involved are:

1. Pick an element, called a pivot from the list. Pivot is usually either the last or the first element in a list.

2. Reorder the list so that all elements in the list, which are smaller than the pivot, are placed before the pivot and so that all elements greater than the pivot are placed after it (equal values can go either way or stay where they were). After this partitioning, the pivot is in its final position (i.e. pivot's position in the final list). This is called the partition operation.
3. Recursively sort the sub-lists of lesser elements and the sub-list of greater elements.

QuickSort works for lists of elements that are numeric as the partitioning algorithm compares the numeric values of elements. It would not work if the elements in the list were movies, as a computer cannot compare movies.

2.3 Partial order based QuickSort

A generic QuickSort would not work in case of comparing non-numeric elements. Hence, the partial order sorting algorithm was developed. The algorithm uses divide and conquer approach of QuickSort algorithm. However, it makes use of partial orders created by humans to generate a total order. This partial order sorting algorithm follows the approach of reCAPTCHA by harnessing human intelligence to sort non-numeric data.

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 pivot is randomly chosen from the list of elements. 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 3 shows the recursion tree for an input list of elements to be sorted.

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.

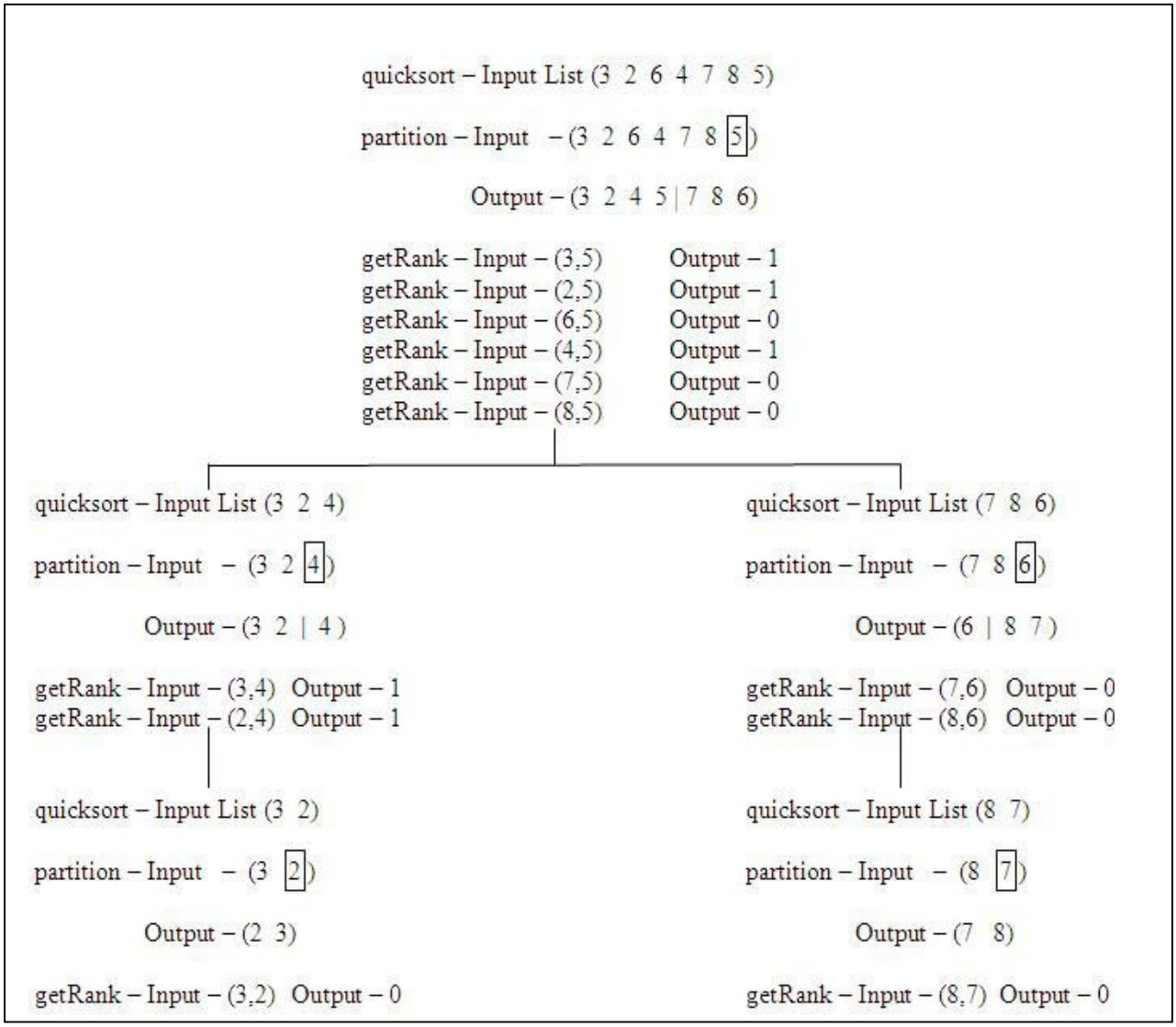

Fig 3. Partial Order QuickSort Example

2.4 Experiments conducted and conclusions drawn

During the course of CS298, experiments were conducted to test the resistance of the algorithm to introduced faults. The section 2.4.1 explains the fault tolerance experiments. The product of the number of partial orders and the partial order set size gives the order

of space complexity of the algorithm. The experiments listed in section 2.4.2 explain the relationship between number of partial orders and partial order set size. The following experiments use the terms listed below:

n – total number of elements

k – number of partial orders

m – partial order set size

2.4.1 Experiments to affirm fault tolerance

Experiment 1: Derive 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).

Rationale: Derive the borderline m that gives back the total order for k = n.

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 (section 2.4.2) 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.

Rationale: Derive the affect of errors for the case of borderline m derived from Experiment 1 above that gives back the total order for k = n.

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. Error introduced vs. Fault Tolerance

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.

2.4.2 Experiments to analyze the space complexity of the algorithm

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

Rationale: Examine the relation between k and m.

For a fixed n = 100, variations of k with respect to m.

(a) k <= n, m <= n

As k increases, m decreases.

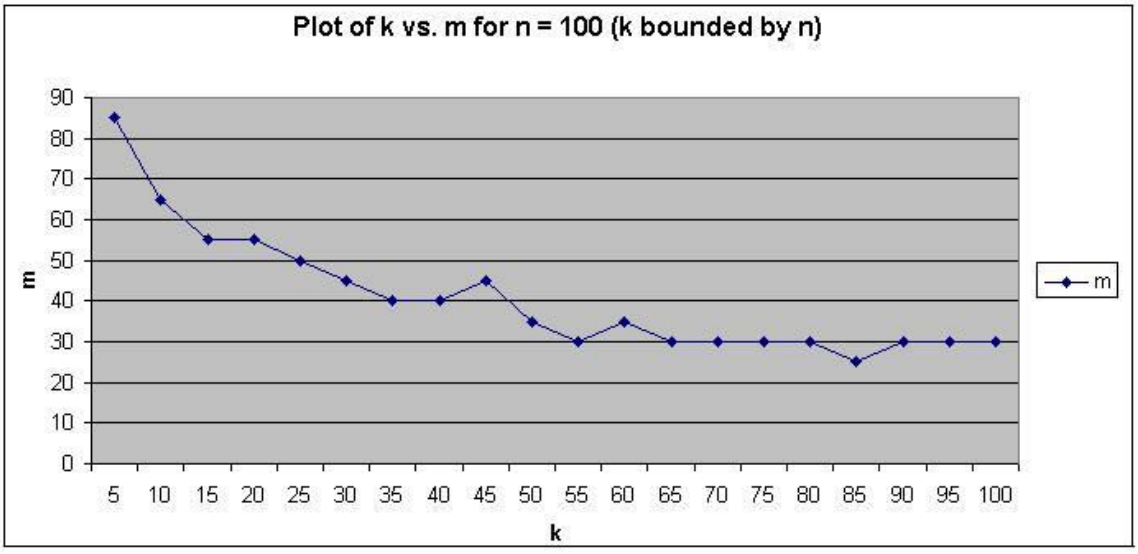

Fig 4. Plot of k vs. m for n = 100 (k bounded by n)

(b) k – unbounded, m <= n

As m increases, k decreases.

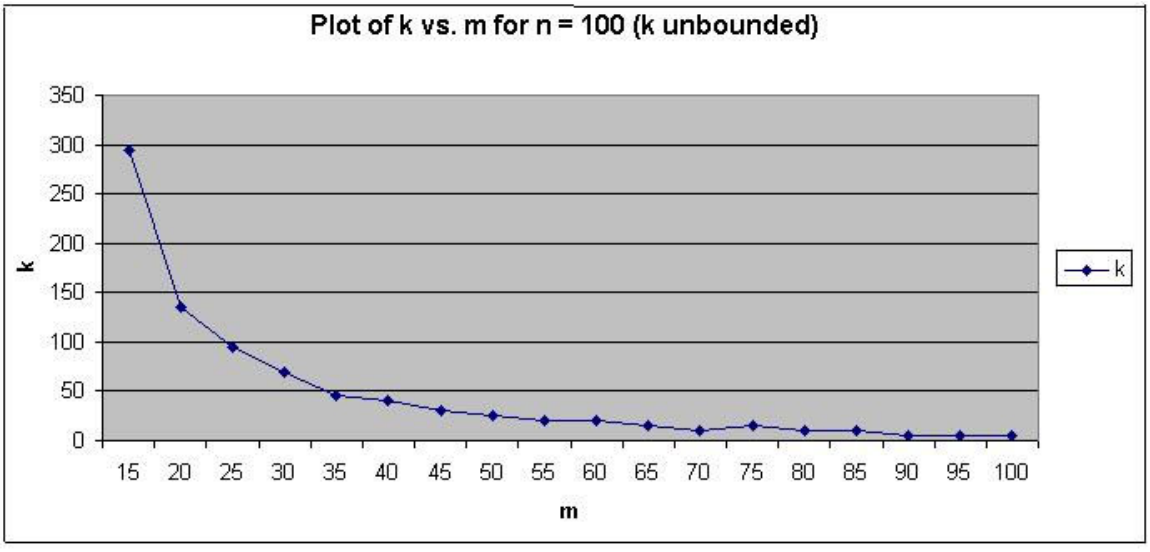

Fig 5. Plot of k vs. m for n = 100 (k unbounded)

Observations:

1. In both the above cases, 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: Arrive at the function relating k (number of partial orders) to n (total number of elements) for a fixed m (partial order set size). 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.

Rationale: Examine the m for k as a multiple of n.

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): Further examine k (number of partial orders) to be a function of f(n) x n (n – total number of elements) 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.

Rationale: Derive the functional dependency of k on n.

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 say 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): Further examine k (number of partial orders) to be a function of n^2 (n – total number of elements) 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.

Rationale: Derive the closest constant relating k with n^2.

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

$k = C \times n^2$

(a) C = 1/4 k = n/4 x n

(b) C = 1/6 k = n/6 x n

(c) C = 1/8 k = n/8 x n

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 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: Examine the case where k (number of partial orders) = n (total number of elements) and tries to derive the borderline m (partial order set size) that would give back the correct total order from the partial orders.

Rationale: Derive the exact m value in the case where k = n.

For different values of n (300 >= n >= 10), the number of partial orders is fixed at n an the exact the partial order size is derived for such a k: k = n

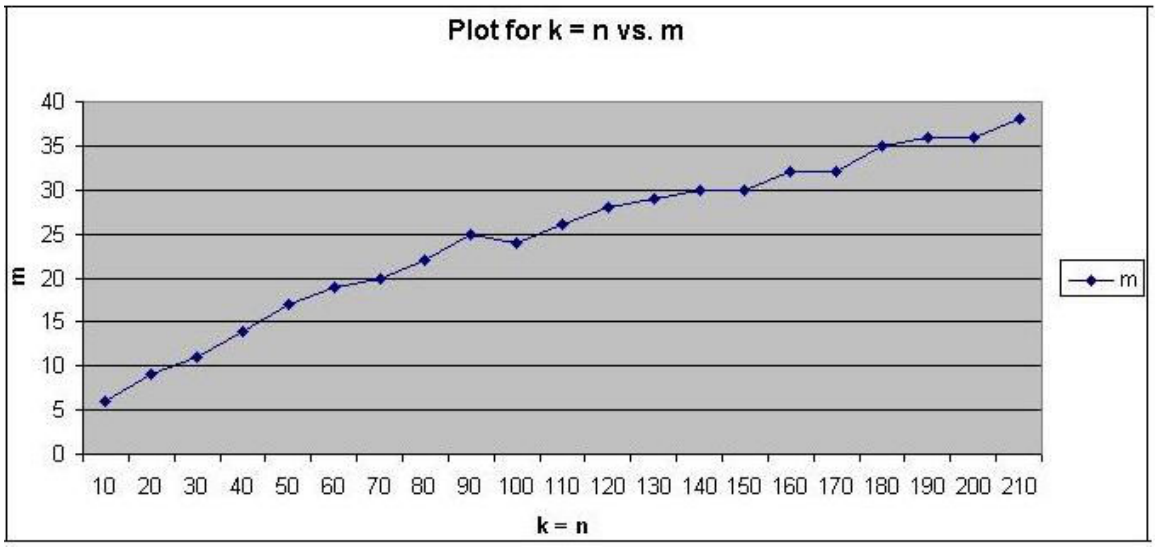

Fig 6. Plot for k = n vs. m

Observations:

1. The partial order set size as a fraction of the 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).

2.4.3 Experiments with random data

Experiment 1: Test the stability of the algorithm over more varied data. Take an input list, generate permutations of the list and generate random partial orders from all these permutations. Now, generate a total order from these partial orders and see if there is a consistent pattern in the output total orders in several runs of the partial order quicksort. The pivot element is randomly chosen.

Rationale: Obtain partial orders from various permutations of the input list and compare the output total orders.

Example run: n = 10, k = 3 x n^2, m = 5, no. of permutations = 3

Input_list:

[213, 784, 152, 856, 212, 394, 157, 208, 838, 511]

Perm_list:

[394, 212, 511, 152, 213, 784, 856, 157, 208, 838]

[213, 856, 784, 212, 208, 511, 394, 152, 838, 157]

[856, 208, 511, 157, 213, 784, 212, 394, 152, 838]

Output_list: 1 [213, 856, 511, 784, 212, 208, 394, 152, 157, 838]

Output_list: 2 [856, 212, 208, 511, 213, 784, 394, 152, 157, 838]

Output_list: 3 [856, 212, 208, 511, 213, 784, 394, 152, 157, 838]

Output_list: 4 [856, 212, 208, 511, 213, 784, 394, 152, 157, 838]

Output_list: 5 [213, 856, 511, 784, 212, 208, 394, 152, 157, 838]

Observations:

The 5 runs of partial order quicksort on the input list give related, but not consistent rankings. For example, the element 838 is the last element in all five total orders. The element 856 comes first in three out of five total orders. When the first element is not 856, the first element is 213 in the other two cases.

Conclusion:

Running quicksort multiple times and taking the item average positions from the outcome must, by the law of large numbers, converge to a stable answer related to the expected position of each element in the underlying probability distribution.

2.4.4 Conclusion of experiments

The conclusion of all the experiments results so far can be listed:

1. The partial order set size is inversely proportional to the number of partial orders to get back an accurate total order.
2. The product of number of partial orders and partial order set size would give us the space complexity.
3. Increasing the number of partial orders would ensure fault tolerance of the algorithm.

3 Software Design

3.1 Tools and Programming Languages

The project has been built using the following languages:

XHTML, Java, JSPs, MySQL

The tools used to develop the test website were:

DreamWeaver 8, Tomcat Server, MySQL Server

The front end has been designed using the tool DreamWeaver 8. The front end consists of XHTML pages. The XHTML code woven with Java has given a lot of JSPs for the front end. The project runs on the Tomcat server. The bean classes have been written in Java. Most of the bean classes involve database querying or database manipulation operations. The database used in this project is MySQL. The connection to the database is established by a method inside the bean class. As all the functions require database querying or manipulation, they are a part of the same bean class. A few JSPs also have Javascript snippets in them.

3.2 Implementation and Design

Algorithm Java Code

The basis of the test website is the partial order QuickSort algorithm that was developed during the course of CS297. The partial order QuickSort algorithm is a variation of the QuickSort algorithm. It includes a function that compares two movies with the help of stored partial orders created by human users. This function is called from the partition

algorithm while comparing two movies. The Java code for the partial order QuickSort is listed below:

```
public static void quicksort(List<ArrayList<Integer>>
Ranked_list, List<Integer> Input_list, int left, int right)
  { Input_list_final = Input_list;
    if (right <= left) return;
    List<Integer> Input_list_new = new ArrayList<Integer>();
    Input_list_new = partition(Ranked_list, Input_list, left,
right);
     int i = partition_index;
    quicksort(Ranked_list, Input_list_new, left, i-1);
    quicksort(Ranked_list, Input_list_new, i+1, right);
  }
```

Listing 1. QuickSort Java Code

```
public static List<Integer> partition (List<ArrayList<Integer>>
Ranked_list, List<Integer> Input_list, int p, int r)
  { int piv = 0, comp = 0, i = 0, j = 0, temp = 0, temp1 = 0,
rank = 1;
    piv = (Integer)Input_list.get(r);
    i = p - 1;
     for(j = p; j < r; j++)
     { comp = (Integer)Input_list.get(j);
       rank = getRank(Ranked_list, piv, comp);

       if(rank == 1)
       { i = i + 1;
         temp = (Integer)Input_list.get(i);
          temp1 = (Integer)Input_list.get(j);
          Input_list.set(i, temp1);
          Input_list.set(j, temp);
       }
     }
     temp = (Integer)Input_list.get(i+1);
     temp1 = (Integer)Input_list.get(r);
     Input_list.set(i+1, temp1);
     Input_list.set(r, temp);
     partition_index = i+1;

     return(Input_list);
  }
```

Listing 2. Partition Java Code

```
public static int getRank (List<ArrayList<Integer>> list, int
piv_elem, int comp_elem)
  { List<Integer> temp_list = new ArrayList<Integer>();
    int flag1 = 0, flag2 = 0, count = 0, count_inv = 0, index_piv
= 0, index_comp = 0;
    for(int r = 0;r != list.size();r++)
     { temp_list = (List<Integer>)list.get(r);
       flag1 = 0;
       flag2 = 0;
       for(int rnk = 0;rnk != temp_list.size();rnk++)
      { int elem = (Integer)temp_list.get(rnk);
         if(elem == piv_elem)
          { index_piv = rnk;
            flag1 = 1;
          }
          if(elem == comp_elem)
          { index_comp = rnk;
            flag2 = 1;
          }
      }

       if((flag1 == 1)&&(flag2 == 1))
       { if(index_comp < index_piv)
         { count++;
          }
          else
          { count_inv++;
          }
       }
     }

     if(count >= count_inv)
     { return(1);
     }
     else
     { return(0);
     }
  }
```

Listing 3. GetRank Java Code

Design

This section lists the software patterns and components that make up the Java based framework. The different components and functions that act on them are given in the table below:

Components	**Functions**
User	register, signin
Movie	add , rank , remove , group, select
Group	create, select, modify, rank
Partial Orders	create, sort

Table 2. Software Components

The relationship between the components can be described from the following figure.

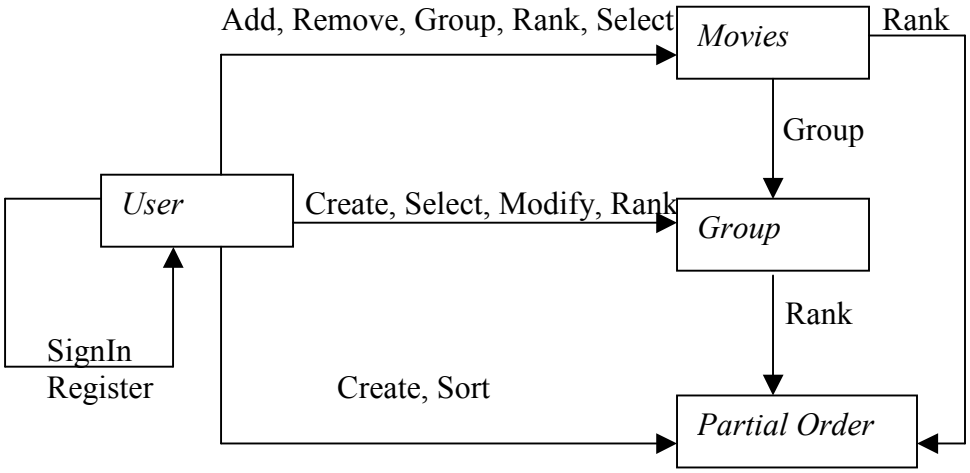

Fig 7. Software entities and relationships

3.3 Database Description

The database used by the project consists of the following tables:

Movie, User, Group, Partial_Orders, Remove

The ER diagram in Fig 8. describes the database:

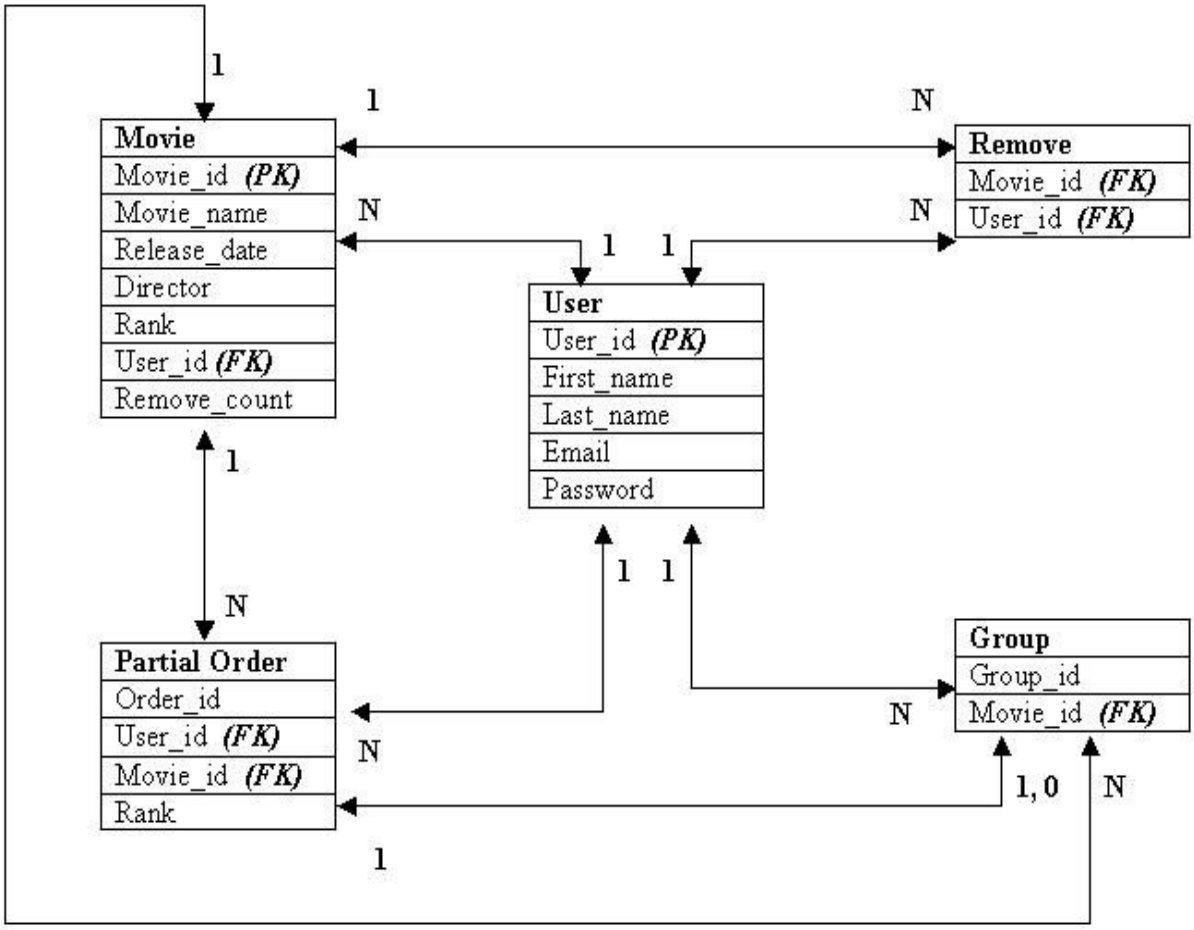

Fig 8. Entity-Relationship Diagram

Movie

Attributes: movie_id, movie_name, release_date, director, rank, added_by_uid, remove_count

The movieid is the primary key in this table which cannot be null. The movieid, rank and remove_count are integers, the remaining fields are all varchar. The reason why the movieid field was chosen to be integer was for the ease of generation of id every time a new movie is added. The movie id is generated by the method in the bean that adds the

movie to the database. It is not entered by the user. The movieid helps to maintain a unique identity of each record in the table.

User

Attributes: user_id, first_name, last_name, email, password

The user_id is the primary key in this table which cannot be null. All the fields are varchar in this table. The users are given the choice of creating user_id or user_name for themselves. However, the user_id created by the user needs to be unique. This aspect is taken care of by the bean while adding the user to the database and storing their personal information. A duplicate user_id entered by a user results in an appropriate message to create and re-enter a new user_id.

Group

Attributes: group_id, movie_id

This table was needed to keep track of the movies in a specific group. The groups are created by users while ranking partial orders. One movie could be in many groups created by many users. This makes it impossible to store the group information in the Movie table as each movie can be a part of more or less than any fixed number.

Partial_Orders

Attributes: order_id, user_id, movie_id, rank

The partial order rankings generated by the users are stored in this table. They are stored as four tuples of attributes given above. For every set of n movies ranked by a user, there are n tuples added to the Partial_Orders table. Each of the n tuples have the same order_id and user_id. This way of storing the partial orders enables flexible size of partial orders i.e. each partial order can have movies sets of various sizes.

Remove

Attributes: movie_id, user_id

This table stores movie_id, user_id tuples. This table helps to keep track of all the movies that every user has marked to delete. It ensures that every movie can be marked for deletion by a user only once so that just one user cannot affect the remove_count in the Movie table and hence lead to its deletion.

3.4 Site flow (How to use 100 best movies!)

This section describes the components and workflow of the test website implementing the partial order QuickSort algorithm. The homepage of the test website 100 best movies executes the partial order algorithm that generates the top 100 movies as ranked by the users and displays the list. Fig 9 shows a screen shot of the 100 best movies homepage.

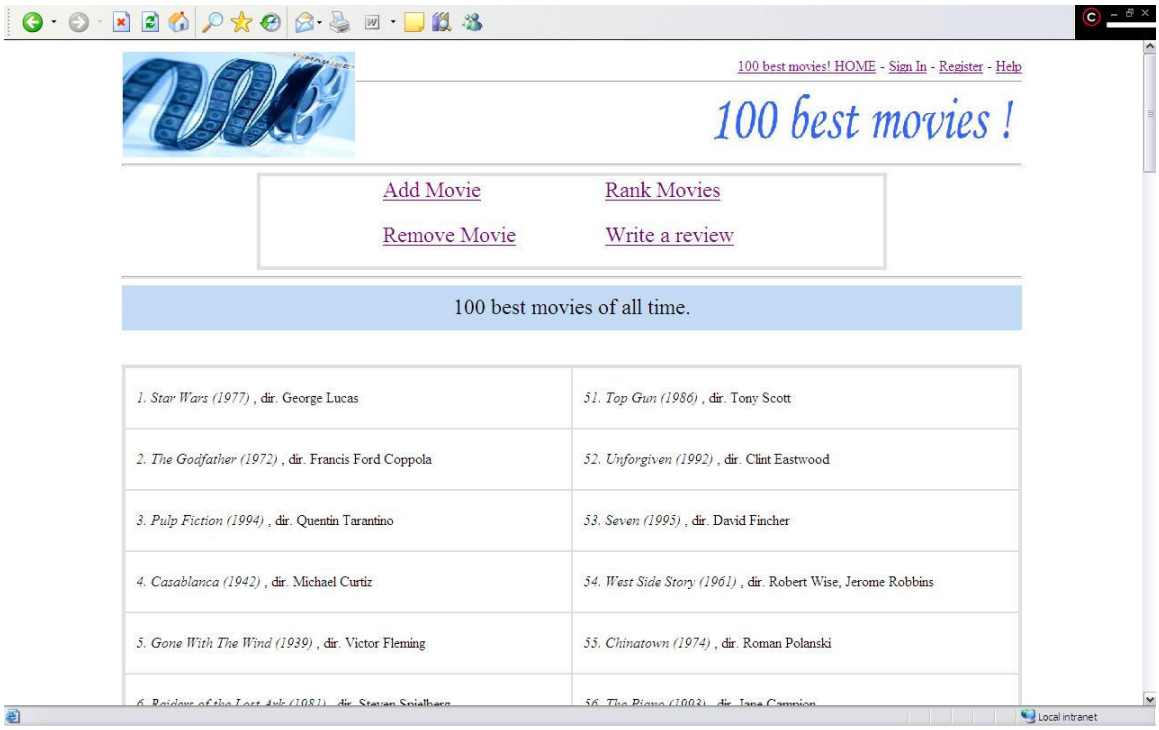

Fig 9. 100 best movies! Homepage

The test website allows the users to add a new movie, rank movies, remove a movie from the list and write a review as shown in Fig 9 above. For any of the described events to take place, a user must be signed in. In case the user is not already signed in clicking on any of the options would take the user to a SignIn page. The user can also chose to sign in by clicking on the link given on the top right corner of the webpage of Fig 9. The SignIn page is shown in Fig 10.

Fig 10. 100 best movies! SignIn

If you are a new user, the test website allows you to register with the website. You can register from the top right corner link on homepage called Register or from the link in SignIn box that says Register Now!. The Register page of the test website is shown in Fig 11.

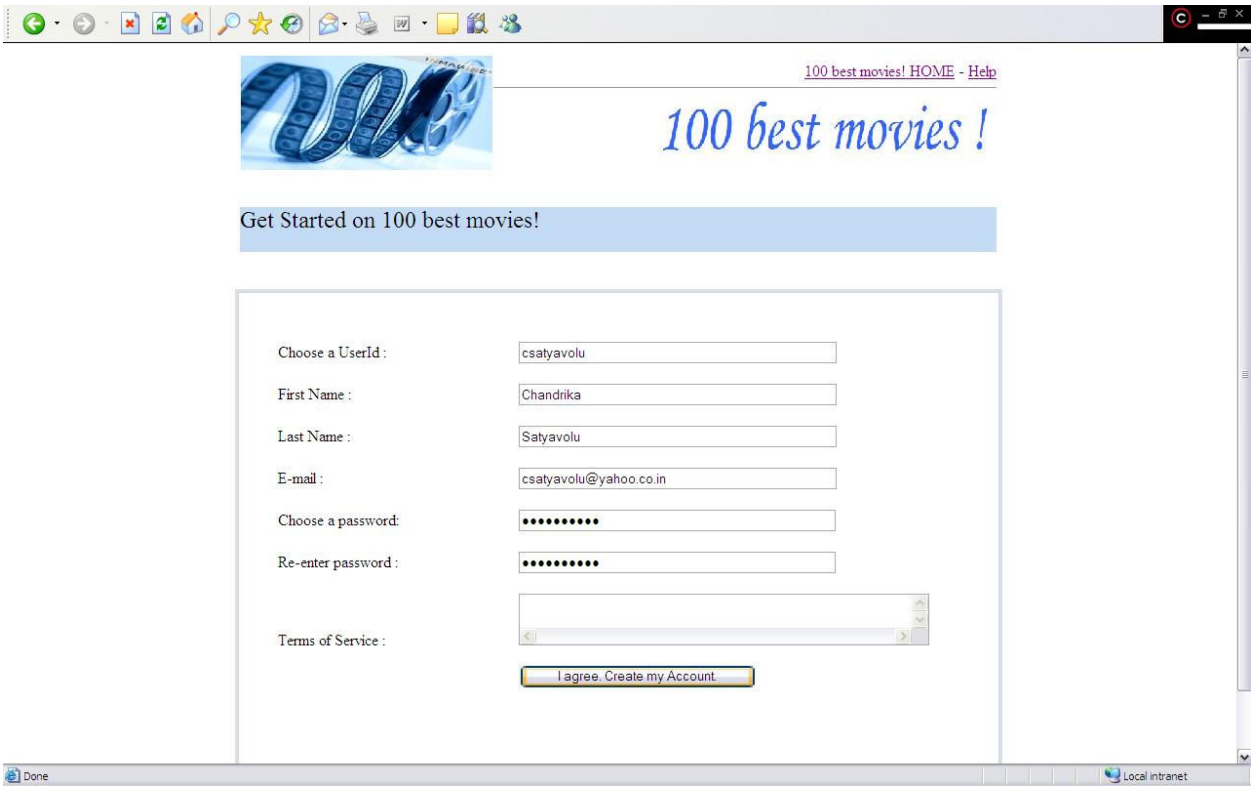

Fig 11. 100 best movies! Register

After you SignIn or Register as a new user, you are taken to your profile page that would display your personal information. It will also display the movies that were added to the database by you along with their total order rank as computed by the partial order QuickSort algorithm. In case you are a new user and you have not added any movies to the database, the My Movies section is blank.

It also allows you to edit your personal information or movie information (only for the movies added by you). It also provides a tab with the links to all the functions that you can do as a registered user of 100 best movies! website. The Profile page is shown in Fig 12.

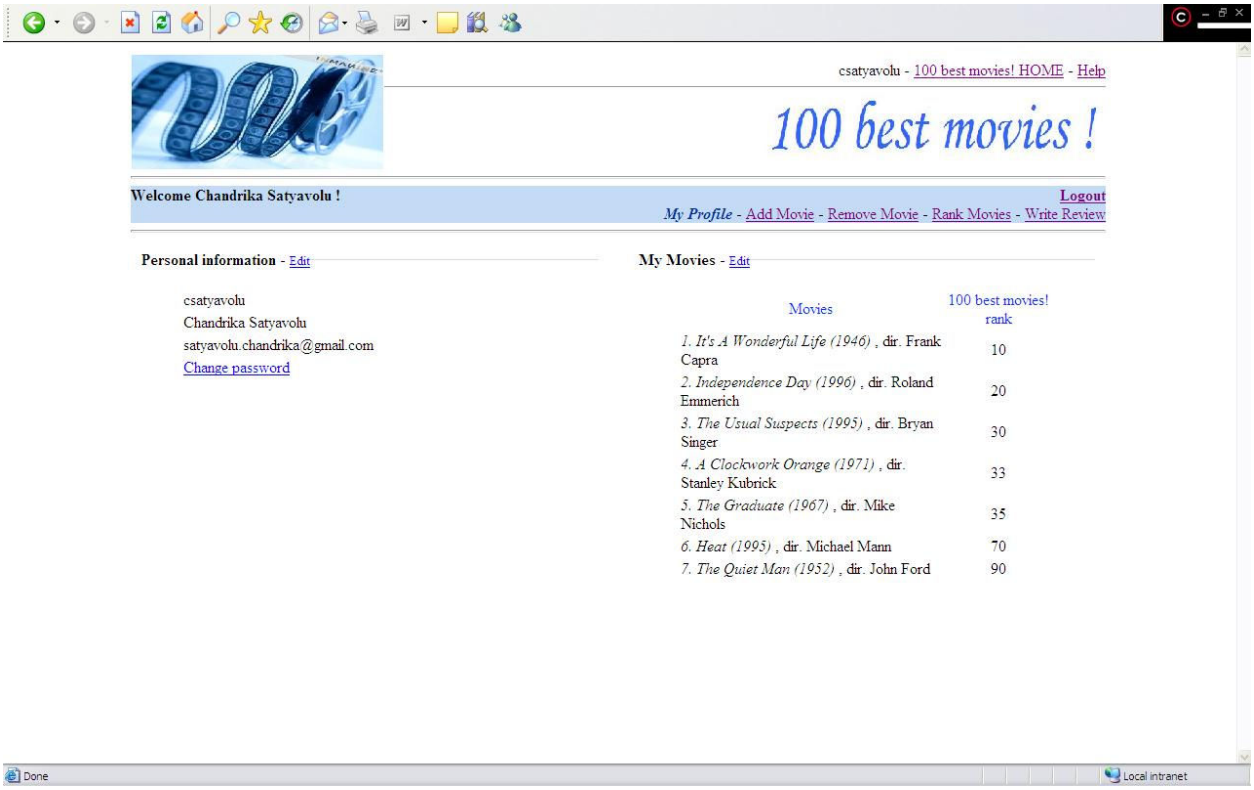

Fig 12. 100 best movies! MyProfile

Once you SignIn and reach your Profile page, you can perform a variety of functions some of them being:

Add Movie, Remove Movie, Rank movies, Write a review

If the user clicks on Add Movie link in the functions tab shown on the Profile page, it opens up the Add Movie page. The user needs to fill in the appropriate movie details and add the movie to the database.

The function allows the movie to be added only if the movie name is not already in the database. The function does not allow for duplication of movies in the database in this manner. The Add Movie page is shown in Fig 13.

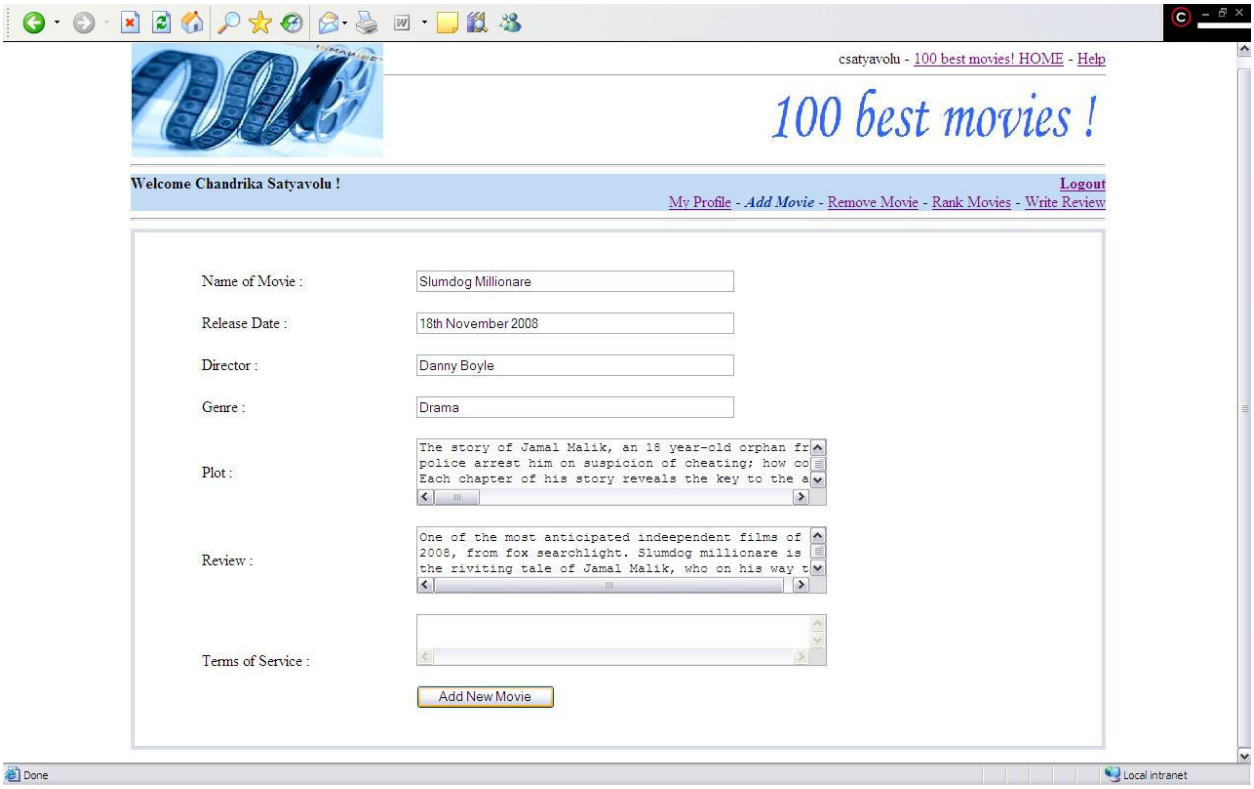

Fig 13. 100 best movies! Add Movie

Another function that the website provides to a signed in user is to Rank Movies. If the user clicks on Rank Movies link in the functions tab of Fig 13., it opens up the Rank Movies page as shown in Fig 14.

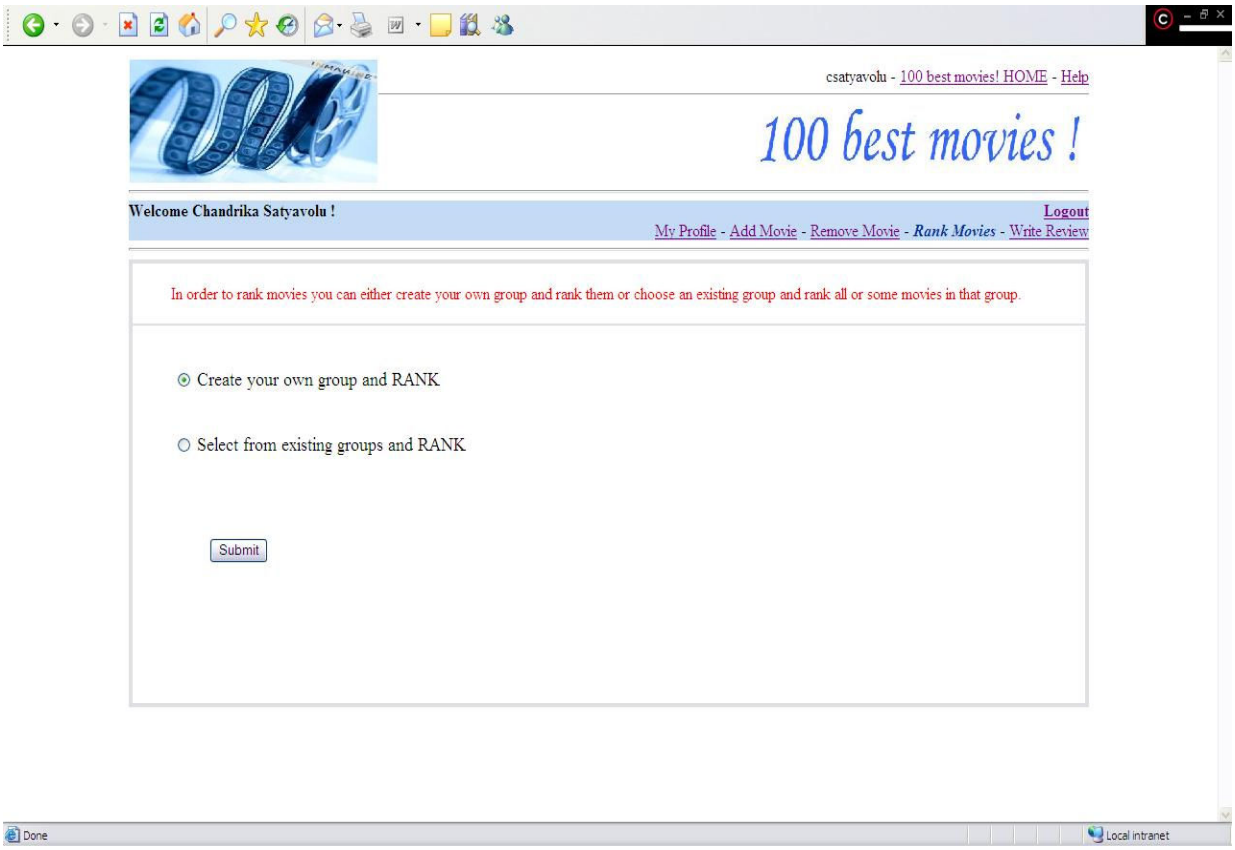

Fig 14. 100 best movies! Rank Movies

The rank movies function in the test website for 100 best movies allows for creating your own group of movies and then rank them or select from existing groups created by other users and then rank them. If you select to create a group, the Create Group page shown in Fig 15 opens up. You can add all the movies of your choice to the group.

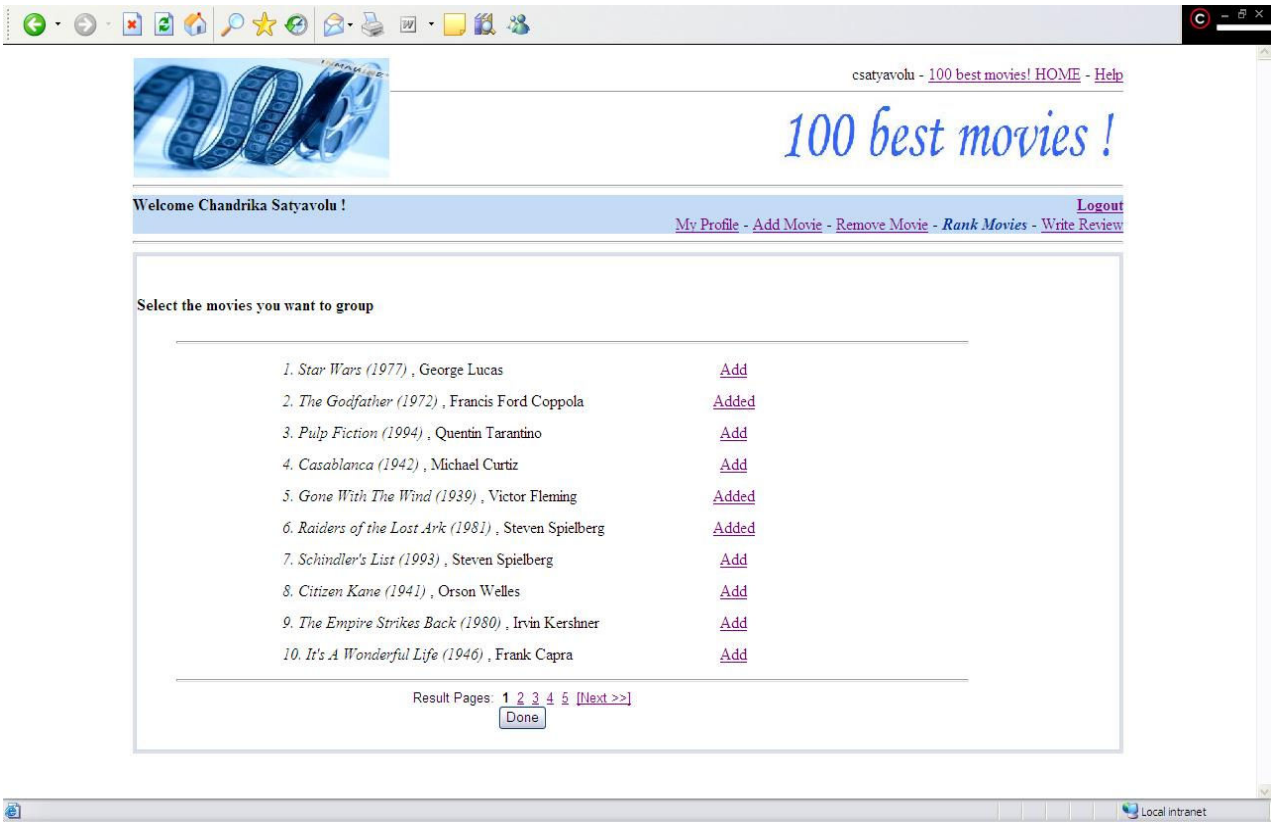

Fig 15. 100 best movies! Create Group

The user can also choose to select from an existing set of groups created by other users. After selecting a group, the user can be faced with the following scenarios:

- Select all movies in existing group and rank them.
- Select some movies from an existing group, make a new group and rank them.
- Select some movies from an existing group and only rank them.
- Select some movies from an existing group, add more movies, make a new group and rank them.
- Select some movies from an existing group, add more movies and only rank them.

Fig 16. shows the page to select a group that comes before any of the scenarios described above.

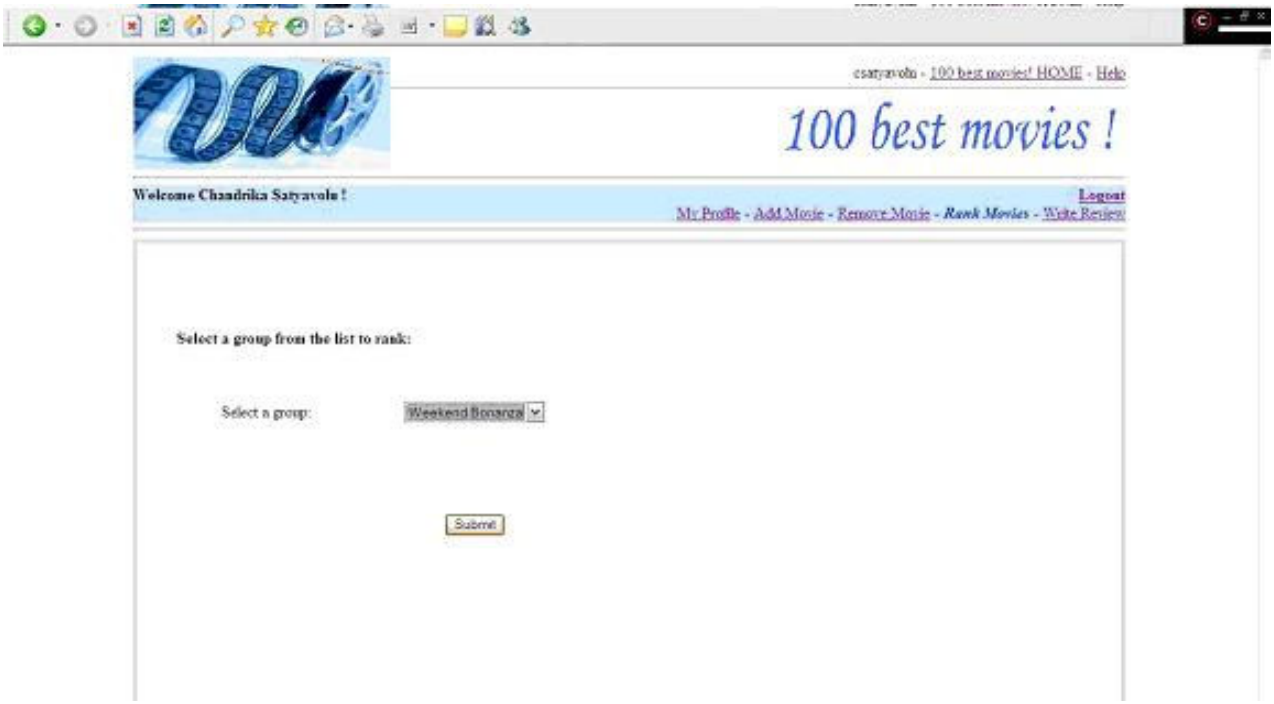

Fig 16. 100 best movies! Select Group

The next screenshot, Fig 17 shows the scenarios (mentioned above) available to a user after a group is selected from a list of pre-existing groups.

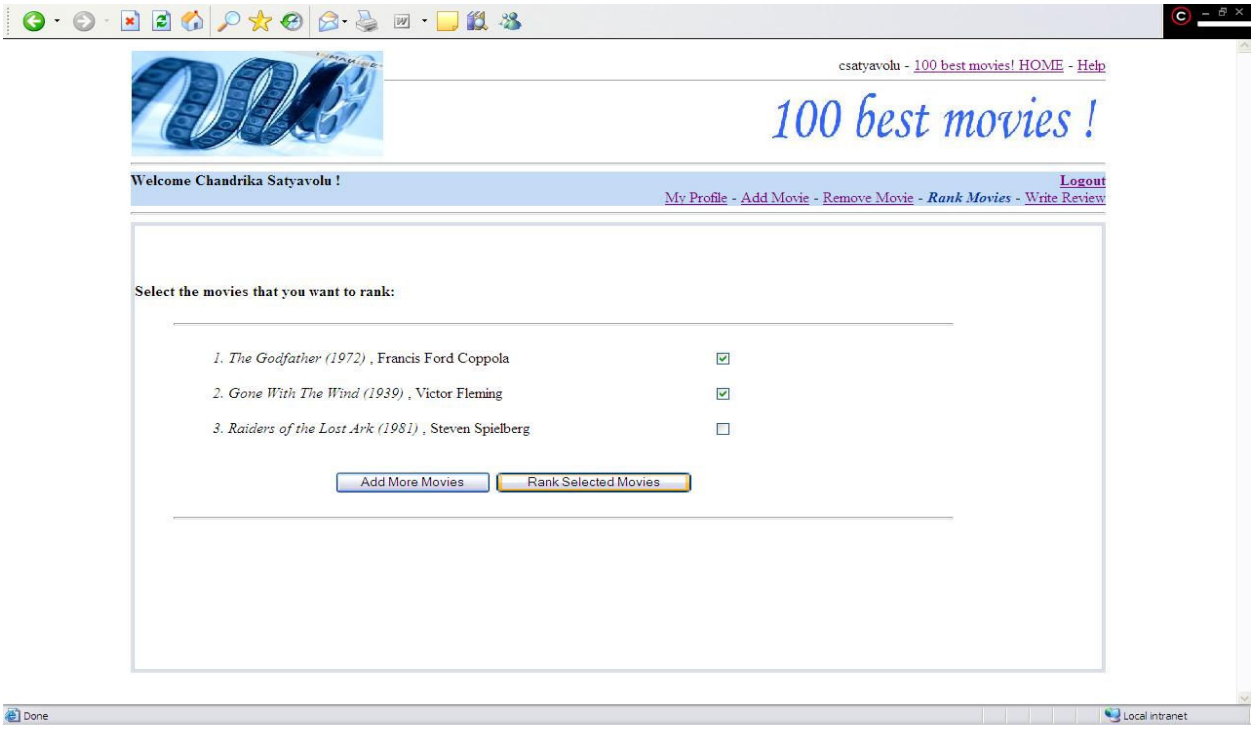

Fig 17. 100 best movies! Select Group

The final stage of creating a new group or selecting from a set of pre-existing groups is shown in Fig 18. This is where the movies are ranked and the partial orders generated by users are stored in the database. The group may or may not be created; it is up to the user. If a group is created, the group name and the movies in the group are stored in the database.

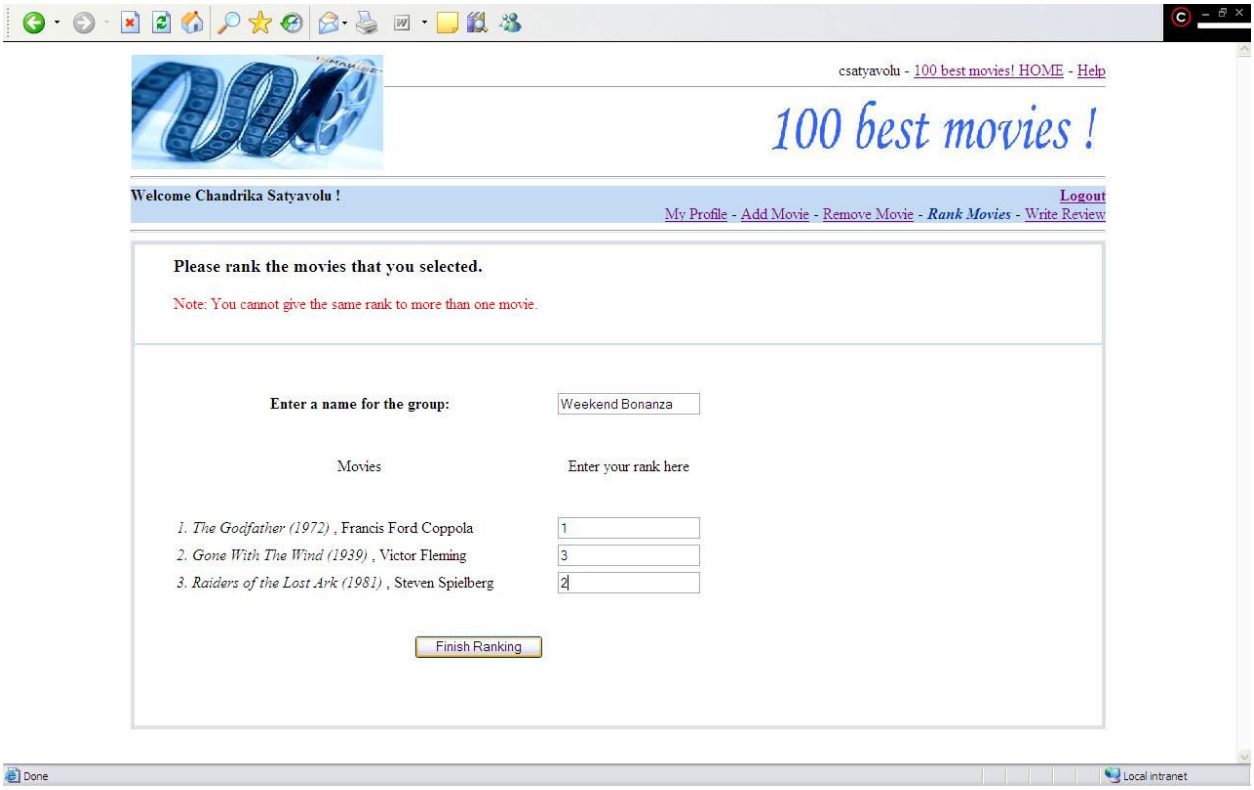

Fig 18. 100 best movies! Rank

The test website also allows for users to remove movies. However, the process of removal is not as simple as the process of adding new movies to the database. The procedure of adding a movie by a user does not depend on other users but removal of a movie does depend on the popularity vote of all users. The users who added a new movie

cannot remove their own movie either. A user can vote to remove a movie only once. This is to make sure that bad user or a computer program does not vote a movie out of the database by repetitive remove function. There is a remove count that keeps track of number of users that requested for a movie to be removed. When this count reaches 100, the movie is automatically removed from the database. The screen shot in Fig 19(a) shows the function to remove a movie.

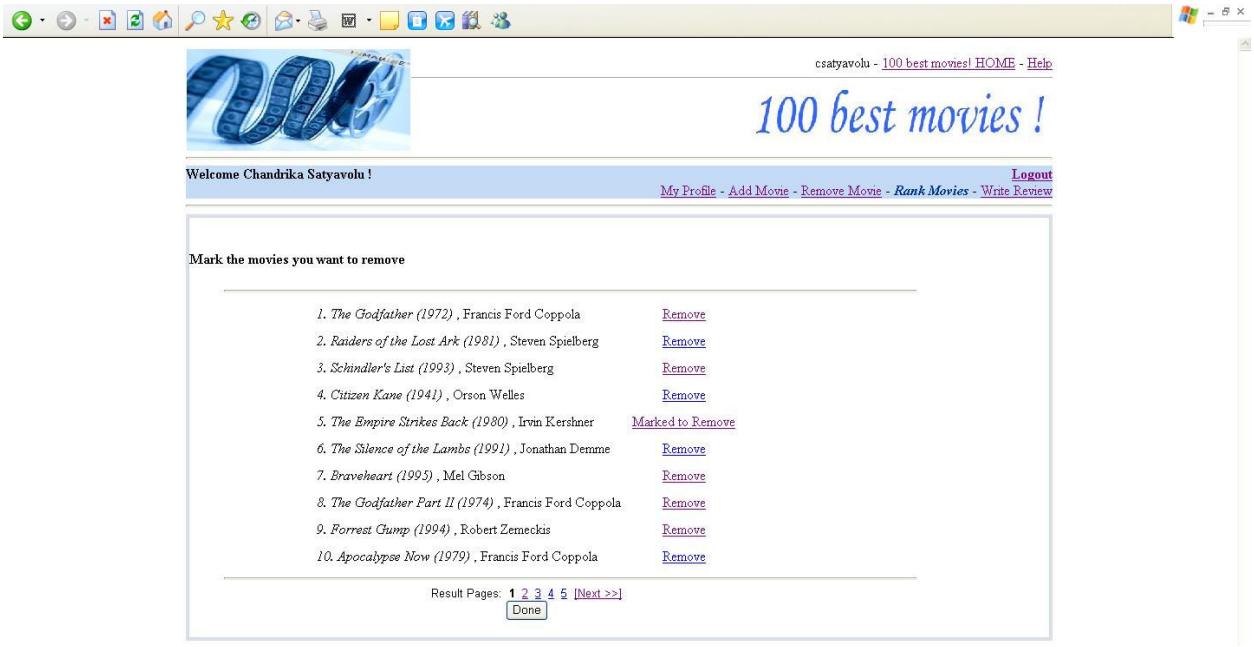

Fig 19(a). 100 best movies! Remove Movie

After the user chooses the movies and marks them to be removed and finally clicks on done, a message appears confirming the same. Also, you can see that once marked the movies do not appear in the list of movies that could be removed for a user as shown in Fig 19(b).

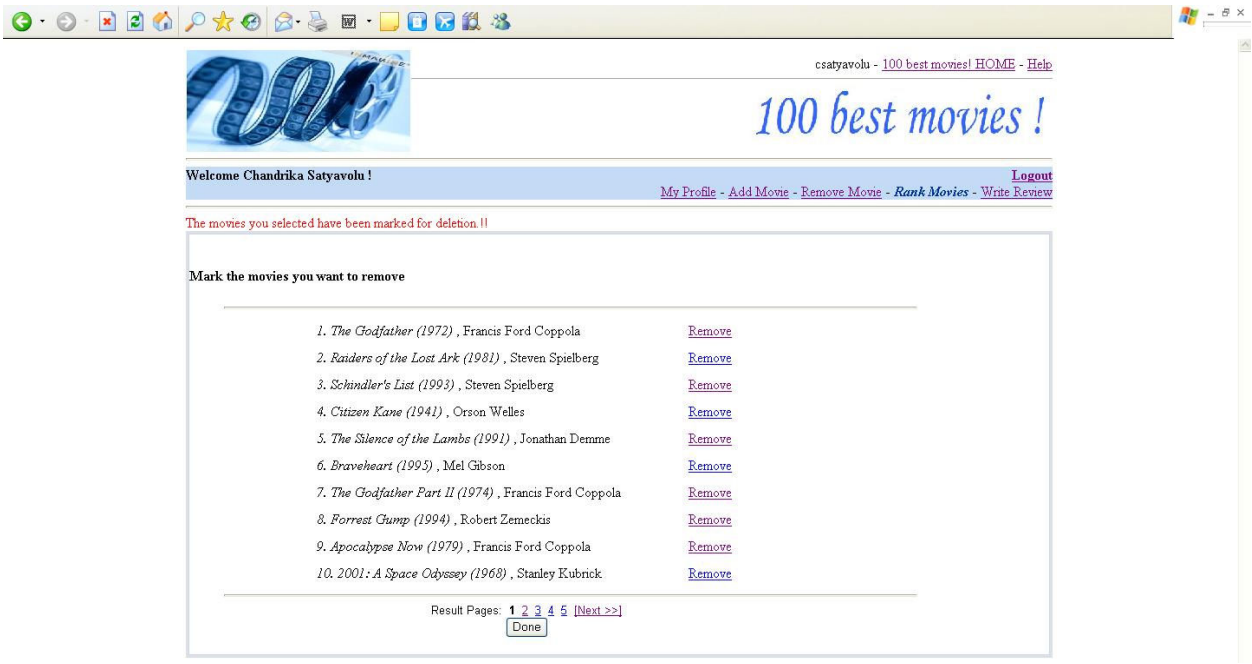

Fig 19(b). 100 best movies! Remove Movie

4 Experiments with users and evolution of system

The test website was tested by users throughout the process of developing the website to help accommodate changes during development. Also, it helped to build a database of movies, users and partial order rankings. The users of the system were mainly my project guide and some friends.

The experiments with actual people using the system helped in the evolution of the website and making it more user friendly. There were a lot of changes made and functions added just because of the unit testing done by users. A few of them were:

Add movie functionality

The add movie functionality more or less remained the same. The only changes were the sizes of the fields that were set keeping in mind the expectations of the users.

Ranking movie functionality

This functionality totally evolved with the way people interacted with the system. The ability to create groups was added on request by users. This gave the way to ranking movies by selecting a group and not just choosing the entire set of movies yourself. This led to making a group while ranking a set of movies necessary. However, after discussing with the project guide, it was finalized to keep the functionality of naming the group and saving the group as optional to the user.

Remove functionality

This functionality was added mainly because of users request to have some amount of control over pushing some movies out of the database. So, it was designed such that a user wanting to remove a movie can only mark it and vote for its removal. The user does not have the absolute authority to remove it from the database.

5 Existing systems

There are two popular systems that I came across that ranked movies. They are:

www.imdb.com

www.rankrz.com

From the imbd webpage listing top 250 movies, I infer that ranking depends on users votes and user ratings. There could be other parameters considered too by the ranking algorithm used by imdb. However, from what the top 250 webpage shows, there is a poll for 250 best movies held by imdb. The movie that gets the largest number of votes and has the maximum popularity gets to be the ranked number one on imbd rankings. The movie that gets the second largest number of votes and has just a little less popularity gets to be the ranked number two on imbd rankings and so on. The user ratings of the movies are also valued in these rankings. These rankings are based on the basis of popularity of a movie. The imdb website showing the top 250 movies is shown in Fig 20 below.

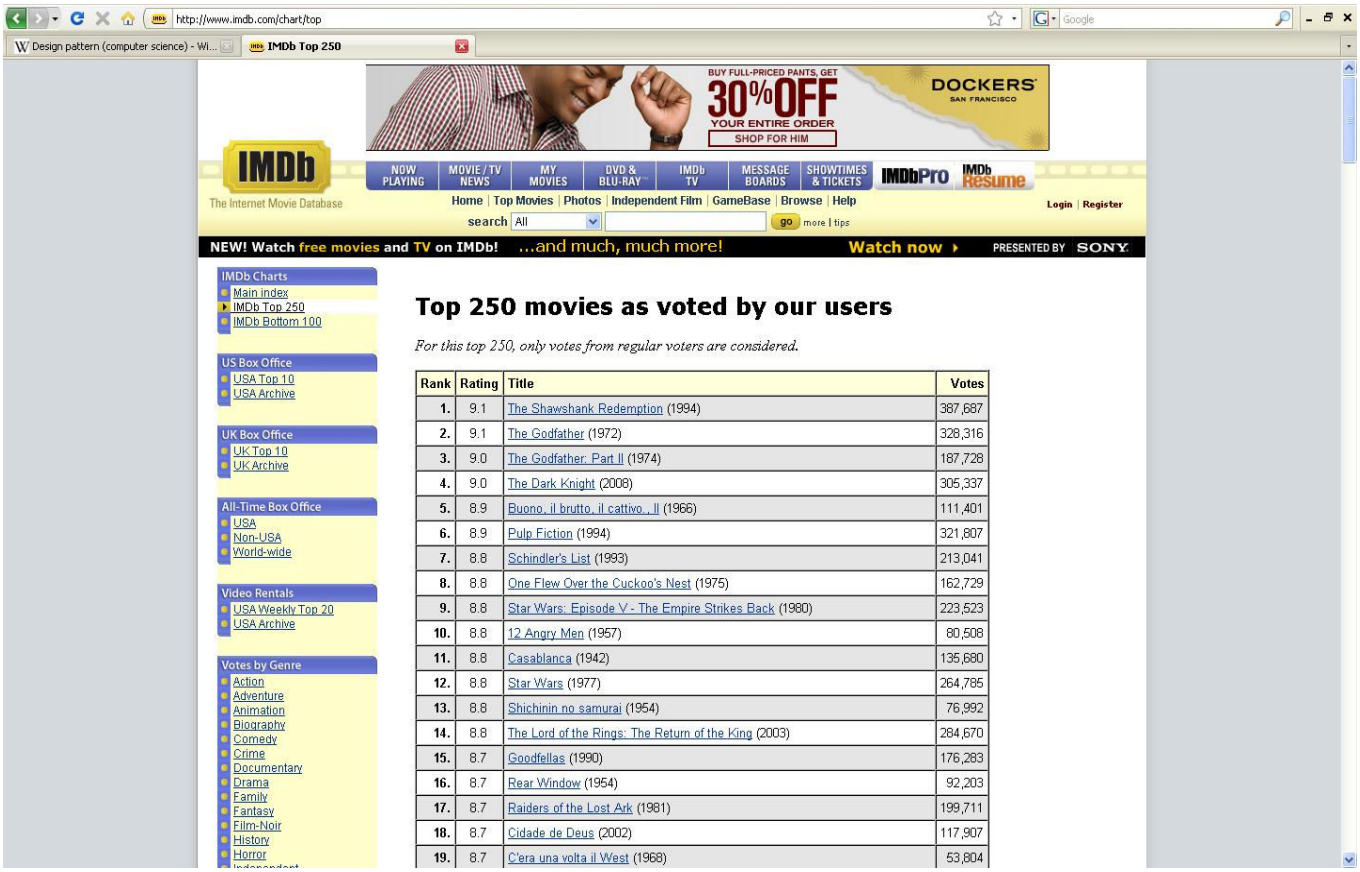

Fig 20. www.imdb.com

Another website that I came across is www.rankrz.com. It generates rankings for many fields like arts, movies, music, books etc. The movies ranking page is shown in Fig 21. This website generates absolute rankings. However, there could be other parameters that are considered too by the ranking algorithm used by imdb. It is similar to the voting system described above with the only difference that a users can select a set of movies that they deem to be the top movies. Finally, the movie that has been selected my most users comes to be the first best movie on the website and so on. Also, if there are movies with same number of votes, they have the same overall ranking. It is based on the popularity of the movies.

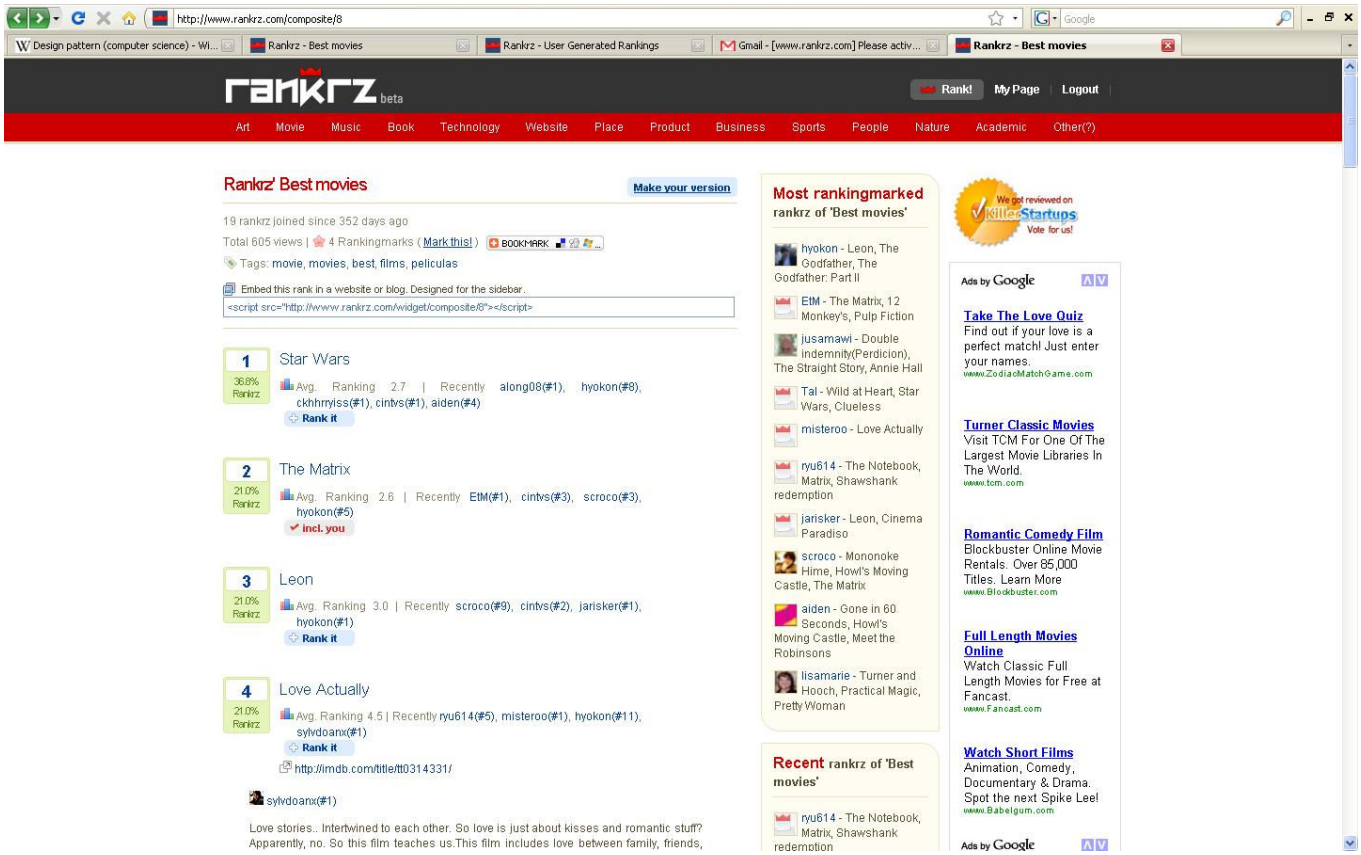

Fig 21. www.rankrz.com

6 Conclusion

We have developed an algorithm that sorts elements that are not possible to be ordered by a computer program. These elements cannot be compared using a computer program. The algorithm uses human inputs in order to derive the total order. We have built a test website: 100 best movies! that uses this algorithm to compare movies and obtain the total order for top 100 movies. We have compared and contrasted our approach with the approach followed by some popular websites like www.imdb.com and www.rankrz.com. The imdb website uses popularity voting and the rankrz website also uses a variation of the popularity voting technique.

This project uses a variation of the quicksort algorithm for total ordering from humanly generated partial orders. We believe that our work can be extended to have the total ordering done by variation of other sorting algorithms too. The performance of using different sort algorithms can be compared and documented.

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
[6] http://en.wikipedia.org/wiki/Quicksort
[7] www.imdb.com
[8] www.rankrz.com