Assignment 3 - COMP SCI 3306

Cyrus Villacampa - 1709135
Rafiqi Rosli - 1682431

1/6/2017

Exercise 1 - *Frequent Itemsets*

1.1 *Solution*:

The simple, randomized(SR) algorithm is implemented as described in section 6.4.1 of the MMDS book. It uses the Park, Chen and Yu(PCY) algorithm to find the frequent sets in a dataset. The algorithm takes a subset from the dataset in random and runs the PCY algorithm on that subset. The found frequent sets are then written in a file called *output.txt*.

1.2 *Solution*:

The Savasere, Omiecinski, Navathe(SON) algorithm is implemented as what is described in the book. It also uses the PCY algorithm to find the frequent sets in a dataset. The implementation of the SON algorithm doesn't use any parallel computing techniques(such as Multithreading or MapReduce) which makes this implementation less efficient. The algorithm starts by dividing the dataset into chunks and uses the PCY algorithm to find all frequent sets in that chunk and all the found sets will be written in a file. Which means that each chunk will generate a file. Afterwards, in between pass, the algorithm will then take the union of all the found frequent sets. In the second pass the algorithm will count how many times each candidate set appears in the whole dataset and filters out the sets that are not found to be frequent(i.e. has support less than the support threshold). The output is then stored in a file called *freq_itemset.txt*.

1.3 *Solution*:

For all cases a sample size of 10% and a support threshold of about 80% of the whole dataset was used.

Input:
`T10I4D100K.dat`
SR output and computation time:
```
No frequent itemset was found of any size
0.15s
```
SON output and computation time:
```
No frequent itemset was found of any size
4.6 seconds
```

Input:
`T40I10D100K.dat`
SR output and computation time:
```
No frequent itemset was found of any size
0.45s
```
SON output and computation time:
```
No frequent itemset was found of any size
7.12 seconds
```

Input:
`chess.dat`
SR output and computation time:
```
Found 21 frequent itemset of size 1
```

```
Found 155 frequent itemset of size 2
Found 615 frequent itemset of size 3
Found 1499 frequent itemset of size 4
0.72s
```
SON output and computation time:
```
Found 19 frequent itemset of size 1
Found 141 frequent itemset of size 2
Found 566 frequent itemset of size 3
Found 1383 frequent itemset of size 4
3min.
```

Input:
`connect.dat`
SR output and computation time:
```
Found 28 frequent itemset of size 1
Found 320 frequent itemset of size 2
Found 2092 frequent itemset of size 3
Found 8938 frequent itemset of size 4
31.67s
```
SON output and computation time:
```
Found 28 frequent itemset of size 1
Found 319 frequent itemset of size 2
Found 2091 frequent itemset of size 3
Found 8962 frequent itemset of size 4
36min.
```

Input:
`mushroom`
SR output and computation time:
```
Found 5 frequent itemset of size 1
Found 9 frequent itemset of size 2
Found 7 frequent itemset of size 3
Found 2 frequent itemset of size 4
0.085s
```
SON output and computation time:
```
Found 5 frequent itemset of size 1
Found 9 frequent itemset of size 2
Found 7 frequent itemset of size 3
Found 2 frequent itemset of size 4
41.11s
```

Input:
`pumsb.dat`
SR output and computation time:
```
Found 25 frequent itemset of size 1
Found 278 frequent itemset of size 2
Found 1756 frequent itemset of size 3
Found 6965 frequent itemset of size 4
```

25.71s

SON output and computation time:

```
Found 25 frequent itemset of size 1
Found 276 frequent itemset of size 2
Found 1760 frequent itemset of size 3
Found 6999 frequent itemset of size 4
20min.
```

Input:

pumsb_star.dat

SR output and computation time:

```
No frequent itemset was found of any size
```

0.36s

SON output and computation time:

```
No frequent itemset was found of any size
```

6.28s

1.4 *Solution*:

For all experiments a support threshold of 80% of the whole dataset was used.

SR(Simple Randomized)

Input \ Samples	1%	2%	5%	10%
T10I4D100K.dat	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size.
T40I10D100K.dat	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size
chess.dat	21, 171, 715 and 1754 frequent itemset of size 1, 2, 3 and 4 respectively	22, 179, 783 and 2113 frequent itemset of size 1, 2, 3 and 4 respectively	19, 149, 625 and 1611 frequent itemset of size 1, 2, 3 and 4 respectively	21, 155, 615 and 1499 frequent itemset of size 1, 2, 3 and 4 respectively
connect.dat	28, 314, 2051 and 8843 frequent itemset of size 1, 2, 3 and 4 respectively	28, 327, 2165 and 9222 frequent itemset of size 1, 2, 3 and 4 respectively	28, 324, 2138 and 9192 frequent itemset of size 1, 2, 3 and 4 respectively	28, 320, 2092 and 8938 frequent itemset of size 1, 2, 3 and 4 respectively
mushroom.dat	5, 9, 7 and 2 frequent itemset of size 1, 2, 3 and 4 respectively	5, 9, 7 and 2 frequent itemset of size 1, 2, 3 and 4 respectively	5, 9, 7 and 2 frequent itemset of size 1, 2, 3 and 4 respectively	5, 9, 7 and 2 frequent itemset of size 1, 2, 3 and 4 respectively
pumsb.dat	25, 277, 1762 and 7071 frequent itemset of size 1, 2, 3 and 4 respectively	25, 279, 1732 and 6940 frequent itemset of size 1, 2, 3 and 4 respectively	26, 275, 1743 and 7001 frequent itemset of size 1, 2, 3 and 4 respectively	25, 278, 1756 and 6965 frequent itemset of size 1, 2, 3 and 4 respectively
pumsb_star.dat	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size

SON(Savasere, Omiecinski, Navathe)

Input \ Samples	1%	2%	5%	10%
T10I4D100K.dat	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size.
T40I10D100K.dat	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size
chess.dat	20, 141, 568 and 1384 frequent itemset of size 1, 2, 3 and 4 respectively	20, 141, 568 and 1384 frequent itemset of size 1, 2, 3 and 4 respectively	20, 141, 568 and 1384 frequent itemset of size 1, 2, 3 and 4 respectively	19, 141, 566 and 1383 frequent itemset of size 1, 2, 3 and 4 respectively
connect.dat	28, 319, 2091 and 8962 frequent itemset of size 1, 2, 3 and 4 respectively	28, 319, 2091 and 8962 frequent itemset of size 1, 2, 3 and 4 respectively	28, 319, 2091 and 8962 frequent itemset of size 1, 2, 3 and 4 respectively	28, 319, 2091 and 8962 frequent itemset of size 1, 2, 3 and 4 respectively
mushroom.dat	5, 9, 7 and 2 frequent itemset of size 1, 2, 3 and 4 respectively	5, 9, 7 and 2 frequent itemset of size 1, 2, 3 and 4 respectively	5, 9, 7 and 2 frequent itemset of size 1, 2, 3 and 4 respectively	5, 9, 7 and 2 frequent itemset of size 1, 2, 3 and 4 respectively
pumsb.dat	25, 276, 1760 and 6999 frequent itemset of size 1, 2, 3 and 4 respectively	25, 276, 1760 and 6999 frequent itemset of size 1, 2, 3 and 4 respectively	25, 276, 1760 and 6999 frequent itemset of size 1, 2, 3 and 4 respectively	25, 276, 1760 and 6999 frequent itemset of size 1, 2, 3 and 4 respectively
pumsb_star.dat	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size	No frequent itemset was found of any size

It can be seen from the results above that there is a really small deviation of the frequent sets found for the SON algorithm, this is because the SON algorithm doesn't take inputs at random, instead it divides the dataset into small chunks. And so even though the sample sizes are different the found frequent sets would almost be the same and sometimes would be exactly the same. The

downside, I think, of the SON algorithm is that you need to implement it using parallel techniques in order for it to be efficient. On the other hand the SR algorithm can be implemented without using any parallel techniques and still be efficient, because it doesn't process the whole dataset, instead it takes a random subset of the whole dataset. The result however wouldn't be as good as the SR algorithm, because it takes a subset from the whole dataset at random then we would expect that the found frequent sets to deviate quite a bit(more than the SON algorithm). However, the found frequent sets of SR algorithm is quite close to the exact numbers. The challenges that we have faced in implementing the SR and SON algorithm is where we implemented one of the algorithms presented in the book in finding frequent sets. We have decided for this exercise to implement the PCY algorithm due to the fact that it doesn't waste unused memory on the first pass and also simple. Another challenging part was in implementing the SON algorithm, this was because the SON algorithm creates a file for each frequent sets found for each chunk and in between pass we had to take the union of all the files and therefore we need to keep track of how many files were created and its content. The second pass of the SON algorithm proved to be challenging because on this pass we had to work with a huge file which was the result of taking the union of the found frequent sets from each chunk and we had to increment their count as we pass through the whole dataset. In this part we had implemented it in a sequential way, not in parallel which we should have done, and so resulted in a runtime that is not as efficient compared to if we have implemented it in parallel(i.e. using Multithreading or MapReduce).

When running the experiments on our implementation we have encountered some problems where our implementation took too long to finish finding all frequent sets. The reason behind this was because we have set the support threshold too small which resulted in finding a lot of frequent sets. And so we have to perform some trial and error in order to determine a support threshold that doesn't make the numbers of found frequent sets blow up. We have also found out that an implementation of Apriori or PCY algorithm is more efficient in terms of runtime than a SON algorithm implemented without using any parallel computing techniques(such as Multithreading or MapReduce) because the SON algorithm tends to do more I/O operations than the other two algorithms. But if parallel computing techniques are employed in the implementation of the SON algorithm then it would outperform the other two.

Exercise 2 - *Clustering*

2.1 *Solution*: Shown at the end of this document

2.2 *Solution*: Shown at the end of this document

Exercise 3 - *Advertising*

3.1 *Solution*:

The worst case in this scenario is that the greedy algorithm always assign a query to an advertiser who bid on that query and has the most number of queries that it has bid on among the other advertisers who has bid on that query. For example for a search query x, advertiser **C** has the most number of queries it has bid on because it has bid on three search queries namely x, y and z, whereas **A** has only one bid and **B** as only 2 bids. This is the case because if the greedy algorithm use up the budget of an advertiser who has bid on a number of search queries then the algorithm would have a small chance of depleting the budget of other advertisers who has only bid

on a few search queries. Therefore the worst case in this situation is when the greedy algorithm assigns all the x queries to **C**(which will deplete its budget) and the algorithm will have no choice but to assign all y queries to **B**(which will also deplete its budget) and therefore leave the z queries unassigned. Resulting only of 4 queries being assigned out of 6 queries.

3.2 *Solution*:

A sequence of queries such that the greedy algorithm will assign as few as half the queries that the optimal offline algorithm would assign is the sequence $xyzz$. This is the case because the optimal algorithm can assign all 4 queries among the advertisers, an example is when the algorithm assign the first query x to advertiser **A**, query y to advertiser **B** and the queries zz to advertiser **C**. However if the algorithm assign both the first and second query(i.e. xy) to advertiser **C** then the queries zz will be unassigned and therefore only half the queries will only be assigned.

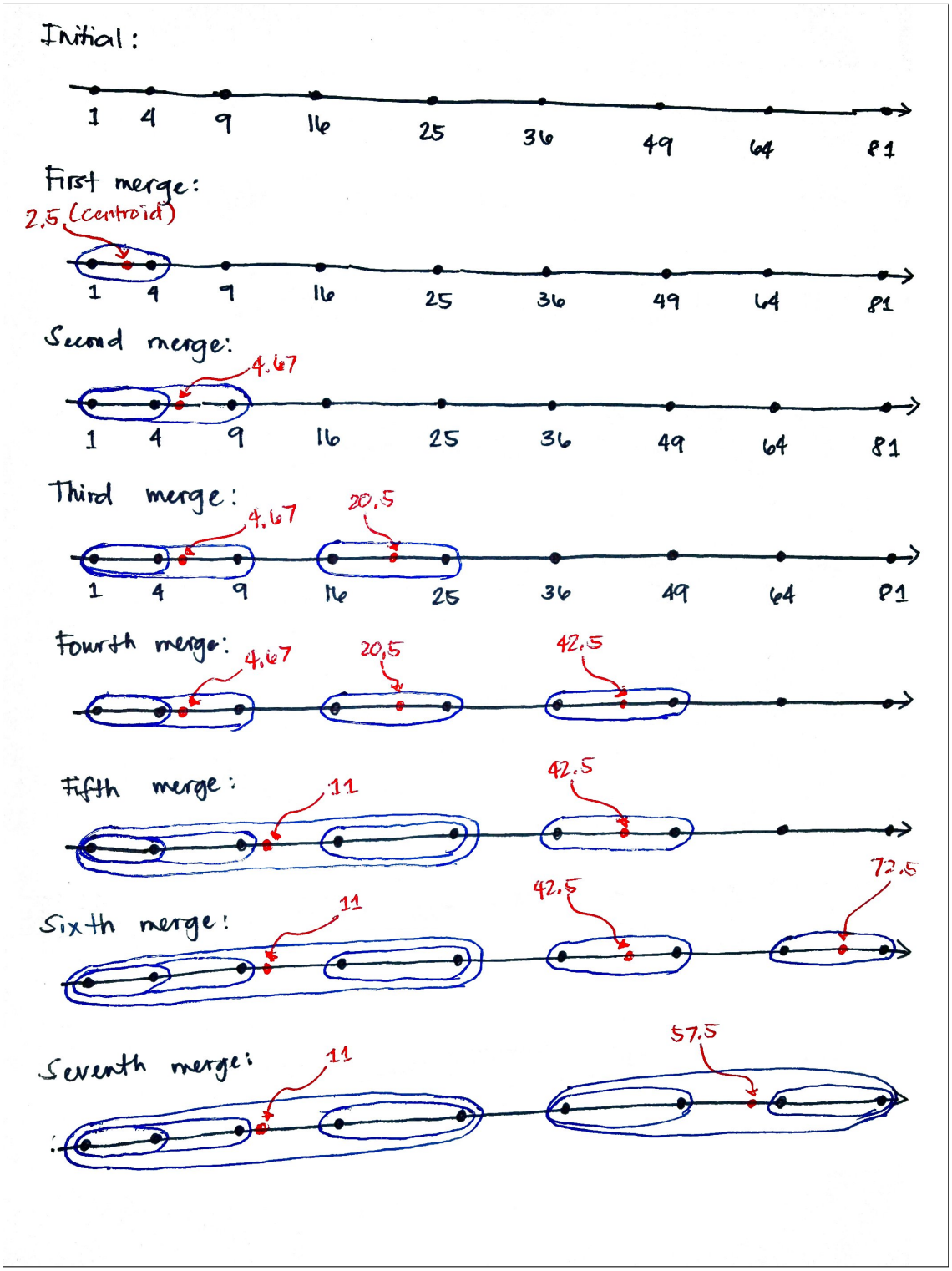

Figure 1: Exercise 2.1 solution

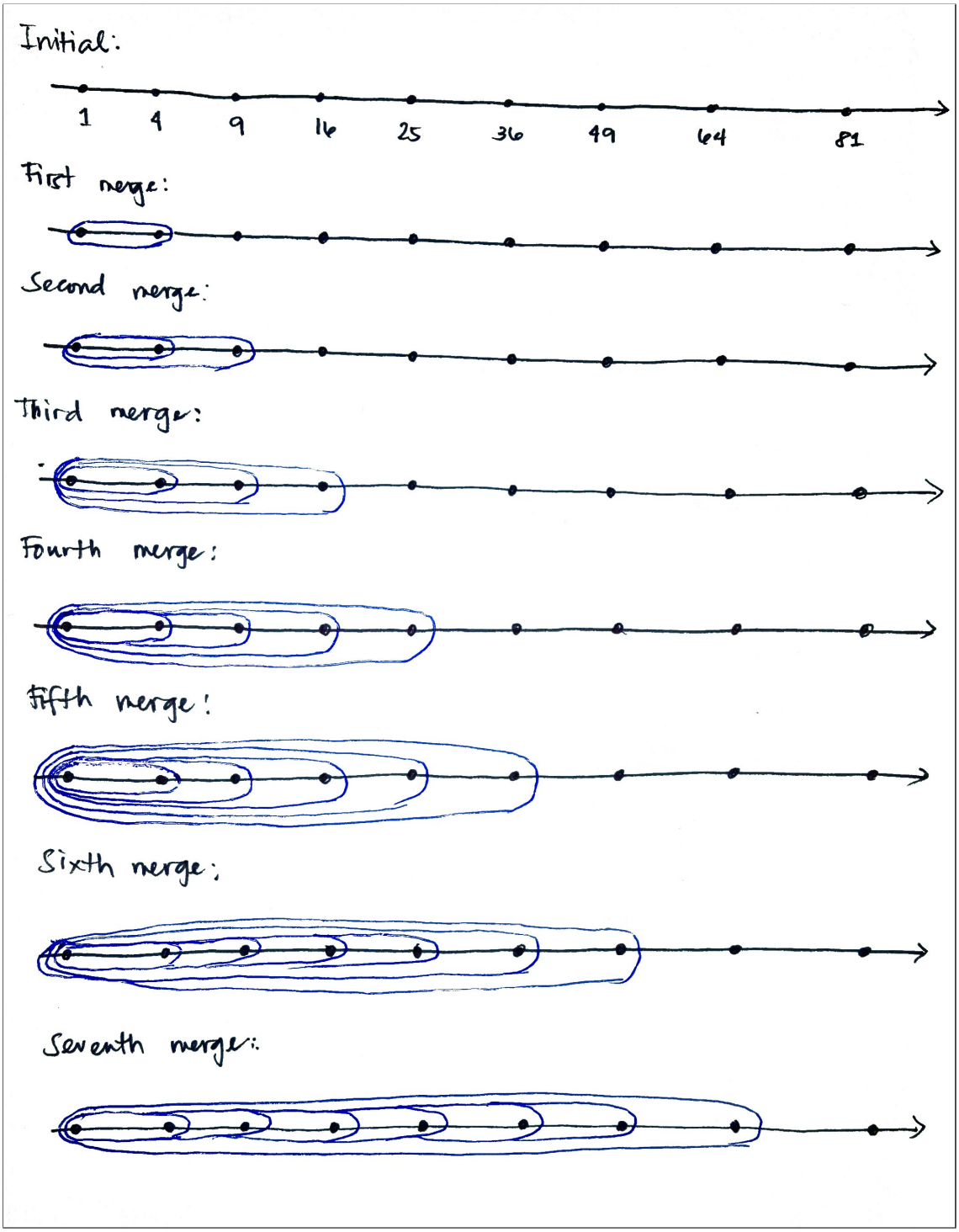

Figure 2: Exercise 2.2 solution