# Mining Frequent Closed Patterns in Microarray Data

Gao Cong, Kian-Lee Tan, Anthony K.H. Tung, Feng Pan
School of Computing
National University of Singapore
3 Science Drive 2, Singapore
{conggao, atung, tankl, panfeng}@comp.nus.edu.sg

## Abstract

*Microarray data typically contains a large number of columns and a small number of rows, which poses a great challenge for existing frequent (closed) pattern mining algorithms that discover patterns in item enumeration space. In this paper, we propose two new algorithms that explore the row enumeration space to mine frequent closed patterns. Several experiments on real-life gene expression data show that the new algorithms are faster than existing algorithms, including CLOSET, CHARM, CLOSET+ and CARPENTER.*

## 1. Introduction

Microarray datasets may contain up to thousands or tens of thousands of columns (genes) but only tens or hundreds of rows (samples). Discovering frequent patterns from microarray datasets is very important and useful, especially in the following: 1) To discover association rules, which can not only reveal biological relevant associations between genes and environments/categories to identify gene regulation pathways but also help to uncover gene networks [1]. 2) To discover bi-clustering of gene expression as shown in [8].

However, these high-dimensional microarray datasets pose a great challenge for existing frequent pattern discovery algorithms. While there are a large number of algorithms that have been developed for frequent pattern discovery and closed pattern mining [3, 4, 7], their basic approaches are based on *item enumeration* in which combinations of items are tested systematically to search for frequent (closed) patterns. As a result, their running time increases exponentially with increasing average length of the records. The high dimensional microarray datasets render most of these algorithms impractical.

It was first shown in [2] that the complete frequent closed patterns can also be obtained by searching in the *row enumeration* space, which was also observed in [5] [1]. Moreover, [2] proposed an algorithm, CARPENTER, to explore the row enumeration search space by constructing projected transposed database recursively.

Considering that many algorithms have been proposed to mine frequent (closed) patterns by item enumeration, it would be interesting to investigate whether some ideas can be borrowed from these algorithms to search row enumeration space more efficiently. In this paper, we developed two new efficient algorithms, RERII and REPT, to explore the row enumeration space to discover frequent closed patterns. Algorithm RERII is inspired by algorithms that mine patterns from vertical layout data [7], while algorithm REPT is inspired by algorithms that are based on FP-tree [4]. But RERII and REPT are very different from them in that both of them adopt row enumeration. Compared with CARPENTER, RERII and REPT use different implementation methods and employ more powerful pruning methods. Several experiments are performed on real-life microarray data to show that the new algorithms are much faster than the existing algorithms, including CLOSET [4], CHARM[7], CLOSET+[6] and CARPENTER[2].

## 2. Problem definition and preliminary

Let $I$=$\{i_1, i_2, .., i_m\}$ be a set of items. Let $D$ be the dataset (or table) which consists of a set of rows $R$=$\{r_1, .., r_n\}$ with each row $r_i$ consisting of a set of items in $I$, i.e $r_i \subseteq I$. Figure 1 shows an example dataset. To simplify notation, in the sequel, we will denote a set of row numbers like $\{r_2, r_3, r_4\}$ as "234". Likewise, a set of items like $\{a, c, f\}$ will also be represented as $acf$. Given a set of items $I'$, the number of rows in the dataset that contain $I'$ is called the **support** of $I'$. A set of items $I' \subseteq I$ is called a **closed pattern** if there exists no $I''$ such that $I' \subset I''$ and the set of rows containing $I''$ is not the same as the set of rows containing $I'$. A set of items $I' \subseteq I$ is called a **fre-**

[1] The submission date of the paper [5] is 1 month after that of [2].

| i | $r_i$ |
|---|---|
| 1 | b,d,e,f |
| 2 | a,c,e,f |
| 3 | a,c,d,e |
| 4 | a,b,c,d,e,g |
| 5 | a,b,c,d,e,f |

**Figure 1. Example Table**

| $i_j$ | rows containing $i_j$ |
|---|---|
| $a$ | 2,3,4,5 |
| $b$ | 1,4,5 |
| $c$ | 2,3,4,5 |
| $d$ | 1,3,4,5 |
| $e$ | 1,2,3,4,5 |
| $f$ | 1,2,5 |
| $g$ | 4 |

**Figure 2. Transposed Table**

**quent closed pattern** if (1) the support of $I'$ is higher than a minimum support threshold, $minsup$; (2) $I'$ is a closed pattern.

**Problem Definition:** Given a dataset $D$ which contains records that are subset of a set of items $I$, the problem is to discover all frequent closed patterns with respect to a user given support threshold $minsup$. In addition, we assume that the database satisfies the condition $|R| << |I|$.

**Preliminary:** CARPENTER is designed based on two basic concepts. One is (projected) transposed table and the other is row enumeration. Table in Figure 2 is a transposed version of table in Figure 1. Let $X$ be a subset of rows. Given the transposed table $TT$, a $X$**-projected transposed table**, denoted as $TT|_X$, is a subset of tuples from $TT$ such that: 1) For each tuple $x$ in $TT$, there exists a corresponding tuple $x'$ in $TT|_X$. 2) $x'$ contains all rows in $x$ with row ids larger than any row in $X$. A complete row enumeration tree on table in Figure 2 is shown in Figure 3.

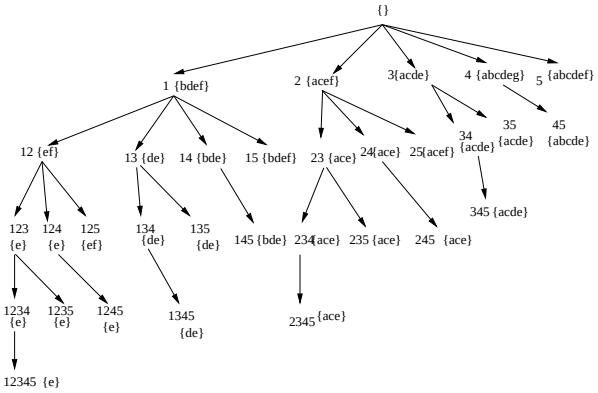



**Figure 3. The row enumeration tree.**

## 3. Algorithm RERII

In RERII, each node $X$ in Figure 3 will be represented with a three-element group $X = \{itemlist, sup, childlist\}$, where $itemlist$ is the closed pattern corresponding to node $X$, $sup$ is the number of rows at the node and $childlist$ is the list of child nodes of $X$. For example, the root of the tree can be represented with $\{\{\}, 0, \{1,2,3,4,5\}\}$ and the node "12" can be represented with $\{\{1,2\}, 2, \{3,4,5\}\}$.

Given a node $X$ in the row enumeration tree, we will perform an intersection of the $itemlist$ of node $X$ with the $itemlist$s of all its sibling nodes after $X$. Each intersection will result in a new node (Note that the intersection may be pruned as discussed later) whose $itemlist$ is the intersection, whose $sup$ is $X.sup + 1$ and whose $childlist$ will be available at next level intersection. And each new node will be intersected with its afterward siblings. In this way, the row enumeration tree will be recursively expanded in a depth-first way.

**Lemma 3.1** *Let $Xr_i$ and $Xr_j$ be two sibling nodes, where $Xr_i < Xr_j$. The following five properties will hold:*

*1) If $Xr_i.itemlist \cap Xr_j.itemlist = \emptyset$, nothing needs to be done.*

*2) If $Xr_i.itemlist = Xr_j.itemlist$, $Xr_j$ will be integrated into $Xr_i$, i.e. $Xr_i.sup = Xr_i.sup + 1$ and any further expansion below $Xr_j$ will be pruned.*

*3) If $Xr_i.itemlist \subset Xr_j.itemlist$, $Xr_i.sup = Xr_i.sup + 1$ and $Xr_j$ will not expand $Xr_i$.*

*4) If $Xr_i.itemlist \supset Xr_j.itemlist$, any further expansion below $Xr_j$ will be pruned and $Xr_j$ will become a candidate extension of $Xr_i$. (Note that whether $Xr_j$ will be a true extension of $Xr_i$ is pending other checking introduced later.)*

*5) If $Xr_i.itemlist \neq Xr_j.itemlist$, $Xr_j$ will become a candidate extension of $Xr_i$.*

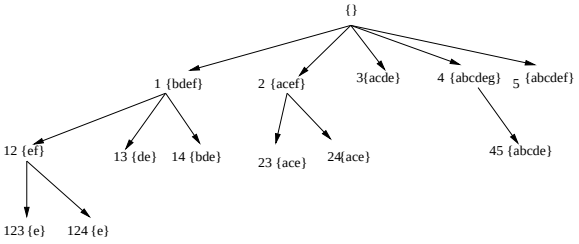



**Figure 4. The pruned row enumeration tree.**

**Example 1** *We now illustrate the Lemma 3.1 with the example table in Figure 1. Suppose minimum support = 1, let us look at how to apply Lemma 3.1 to prune the complete row enumeration tree shown in Figure 3. Consider node 1, its itemlist is a subset of that of node 5 (case 2) while the intersection of its itemlist with the others satisfies the case 5. As a result, we increase the support of node 1 by 1 and extend node 1 with nodes 2, 3 and 4 to get three child nodes. Next we process the node 12, the intersection of the itemlist*

*of 12 with the itemlist of 13 and 14 satisfies the case 5 and we extend 12 with 13 and 14. At the node 123, the intersection of its itemlist with that of 124, i.e. e satisfies with case 2. In this way, we get a close pattern $\{e\}$ with support=4. Next we proceed to node 13 and the intersection of its itemlist with that of node 14, i.e. $\{de\}$, satisfies case 3. Thus we get a closed pattern $de$ with support =3. The extension is done in a depth-first way. The nodes that are actually checked are shown in the Figure 4.*

**Algorithm RERII(D, minsup)**

1. Scan database $D$ to find the set of frequent items $F$
2. Remove the infrequent items in each row $r_i$ of $D$
3. Each $r_i$ forms a node in the first level of row enumeration tree and let $N$ be the set of nodes
4. RERIIdepthfirst($N, FCP$)
5. Let $CF$ be the set of closed items in $F$, $FCP$ = $FCP \cup CF$ and return $FCP$

**Procedure: RERIIdepthfirst(N, FCP)**

6. **for** each node $n_i$ in $N$
7. $N_i$ = null
8. **if** the left row enumeration cannot be frequent **return**
9. **for** each $n_j$ in $N$, where $n_j > n_i$
10. compute the frequency of items to do support pruning
11. d = $n_i.itemlist \cap n_j.itemlist$
12. **if** $|d| > 1$
13. **if** $n_i.itemlist = n_j.itemlist$
14. remove $n_j$ from $N$
15. increase $n_i.sup$ and $n'.sup$ ($n' \in N_i$) by 1
16. **if** $n_i.itemlist \subset n_j.itemlist$
17. increase $n_i.sup$ and $n'.sup$ ($n' \in N_i$) by 1
18. **if** $n_i.itemlist \supset n_j.itemlist$
19. remove $n_j$ from $N$
20. **if** $n_i.itemlist \cup d$ is not discovered before
21. add $n'$ ($n'.sup = n_i.sup + 1, n'.itemlist = d$) to $N_i$
22. **if** ($n_i.itemlist \neq n_j.itemlist$)
23. **if** $n_i.itemlist \cup d$ is not discovered before
24. add $n'$ ($n'.sup = n_i.sup + 1, n'.itemlist = d$) to $N_i$
25. **end for**
26. **if** $n_i.sup \geq minsup$, add $n_i.itemset$ to $FCP$
27. **if** $N_i \neq \emptyset$ and $N_i$ satisfies support pruning
28. call RERIIdepthfirst($N_i$, $FCP$)
29. **end for**

**Figure 5. Algorithm RERII**

We give the pseudo code of algorithm RERII in Figure 5. We further optimize the algorithm RERII using three techniques that will be explained as follows.

**Single Item Pruning.** RERII has already discovered the set of frequent single items by scanning the database once, we only need to discover those frequent closed patterns longer than 1. Therefore, if RERII finds that an enumeration node cannot result in pattern longer than 1, that node will be pruned. Algorithm RERII applies such an optimization at line 3 and line 12.

**Support Pruning.** RERII tries to utilize support pruning at three levels.

*Level 1.* This pruning is done at line 8 of RERIIdepthfirst(). Given a node $X$ with $k$ child nodes $Xr_1, Xr_2, ..., Xr_k$, for any child node $Xr_i$, if $Xr_i.sup + k - i < minsup$, there is no need to do any further enumeration below node $Xr_i$.

*Level 2.* This pruning is done at line 10 of RERIIdepthfirst(). Given a node $X$ with $k$ child nodes $Xr_1, Xr_2, ..., Xr_k$, for any child node $Xr_i$, we compute the supports for items in $X.itemlist$ (= $i_1, i_2, ...i_m$) in all nodes $Xr_j$ such that $i \leq j \leq k$. The counter $support(i_l)$ for each item $i_l$ in $X.itemlist$ is initialized with $X.sup$ and will be increased by 1 if the item is in $Xr_j.itemlist$. On the basis of *single item pruning*, we only need to discover patterns longer than 1. We can derive the following two pruning methods. First, if there are fewer than two items such that $i_l \in Xr_i$ and $support(i_l) \geq minsup$, there is no need to do any further enumeration below node $Xr_i$. Second, if there are fewer than two items such that $i_l \in X.itemset$ and $support(i_l) \geq minsup$, there is no need to do any further enumeration below nodes $Xr_j$ ($i \leq j \leq k$).

*Level 3.* This pruning is done at line 27 of RERIIdepthfirst() after $N_i$ is filled, i.e. the child nodes of $n_i$ are obtained. The detailed approach is similar to that in Level 2.

**Redundant Pruning.** On the basis of the lemma 3.1, at a node $X$, if pattern $X.itemlist$ has already been discovered in an earlier enumeration, we can prune node $X$ and any further enumerations below $X$.

## 4. Algorithm REPT

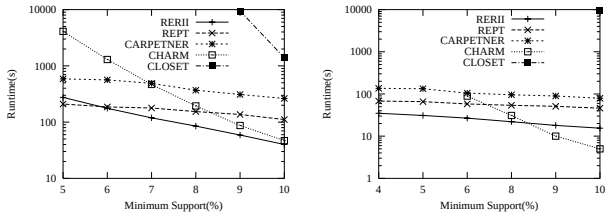
Like CARPENTER, algorithm REPT traverses the row enumeration tree with the help of projected transposed table. Its first main difference from CARPENTER is that REPT represents (projected) transposed table with prefix trees, which can help in saving memory and saving computation in counting frequency. The second main difference of REPT from CARPENTER lies in pruning method. The prefix tree used to represent transposed table is similar to the FP-tree used in [4] to represent original table. In FP-tree, each node represents an item while the node of prefix tree used in REPT represents a row. The algorithm details are ignored here because of space limitation.

## 5. Performance Studies

In this section, we will evaluate RERII and REPT in terms of both the efficiency and memory usage. All our experiments were performed on a PC with a Pentium IV 2.4 Ghz CPU, 1GB RAM running Linux and a PC with Pentium IV 2.6, 1 G RAM running Windows XP. Algorithms were coded in Standard C. We compare RERII and REPT

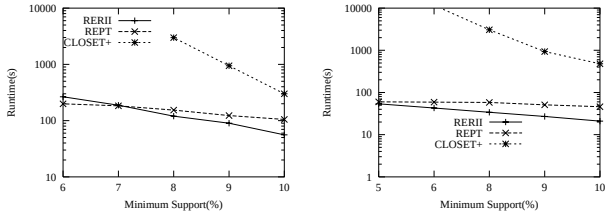
against three other closed pattern discovery algorithms [2], CARPENTER [2], CHARM [7] and CLOSET [4] on Linux and against CLOSET+ [6] on Windows.

Our experiments are performed on 2 real-life datasets, which are the clinical data on breast cancer (BC) [3] and ALL-AML leukemia (ALL) [4]. In the BC dataset, there are 97 tissue samples and each sample is described by the activity level of 24481 genes. In the ALL dataset, there are 72 tissue samples each described by the activity level of 7129 genes. The datasets are discretized by doing a equal-width partition for each column with 50 buckets.




(a) Breast Cancer (b) ALL-AML leukemia

**Figure 6. Runtime Performance**




(a) Breast Cancer (b) ALL-AML leukemia

**Figure 7. Comparison with CLOSET+**

Figure 6 shows the experimental results on our three datasets. Note that the y-axes of these graphs are in logarithmic scale. At some points in Figure 6(b), the runtime of CHARM is not shown because CHARM cannot finish by reporting error after using up all available memory. We do not give the runtime of CLOSET on all points because it is too slow and showing them will make the differences in runtime of other algorithms unclear in these graphs. Among the five algorithms, we find that RERII is usually the fastest while CLOSET is the slowest and has the steepest increases in run time as $minsup$ is decreased. CHARM is generally 1 order of magnitude slower than RERII and REPT at low support. Compared to CARPENTER, RERII is usually 2-4 times faster and REPT is usually 1-2 times faster. These results clearly show that the two proposed algorithms in this paper are efficient.

Figure 7 shows the comparison results with CLOSET+. CLOSET+ cannot finish by reporting error after using up all available memory at $minsup = 7\%$ on BC. Figure 7 shows that both RERII and REPT are usually 1 order of magnitude faster than CLOSET+.

In our experiments, we also observe the memory usage, which usually follows the following relation: CHARM $>$ RERII $>$ CLOSET, CARPENTER, CLOSET+ $>$ REPT. REPT needs the least memory while CHARM is the most memory consuming. It is also interesting to note that tree-based schemes (e.g., CLOSET, REPT using FP-tree or prefix tree) generally consume less memory, while non-tree-based algorithms(e.g., CHARM, RERII) are typically more efficient on the data that we use.

## 6. Conclusions

In this paper, we have proposed two new algorithms, RERII and REPT, to discover frequent closed patterns. Several experiments showed that the proposed algorithms are faster than existing algorithms, including CLOSET, CHARM, CLOSET+ and CARPENTER.

## References


[1] C. Creighton and S. Hanash. Mining gene expression databases for association rules. *Bioinformatics*, 19, 2003.

[2] F. Pan, G. Cong, A. K. H. Tung, J. Yang, and M. J. Zaki. CARPENTER: Finding closed patterns in long biological datasets. In *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining(KDD)*, 2003.

[3] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. 7th Int'l Conf. Database Theory (ICDT)*, 1999.

[4] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proc. ACM-SIGMOD Int'l Workshop Data Mining and Knowledge Discovery (DMKD)*, 2000.

[5] F. Rioult, J.-F. Boulicaut, B. Cremileux, and J. Besson. Using transposition for pattern discovery from microarray data. In *Proc. ACM-SIGMOD Int'l Workshop Data Mining and Knowledge Discovery (DMKD)*, 2003.

[6] J. Wang, J. Han, and J. Pei. CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, 2003.

[7] M. J. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed association rule mining. In *Proc. SIAM Int'l Conf. on Data Mining (SDM)*, 2002.

[8] Z. Zhang, A. Teo, B. Ooi, and K.-L. Tan. Mining deterministic biclusters in gene expression data. In *4th Symposium on Bioinformatics and Bioengineering*, 2004.


[2] We are grateful to Dr. Mohammed Zaki for the Linux version source code of CHARM and Dr. Jiawei Han and Dr. Jianyong Wang for the executable code of CLOSET+ running on Windows

[3] http://www.rii.com/publications/default.htm

[4] http://www-genome.wi.mit.edu/cgi-bin/cancer