$Hash^2$ GSP

Qiang Li Keren Zhou Gang Zeng Wuzhao Zhang
Institute of Computing Technology
Chinese Academy of Sciences

December 10, 2014

1 Introduction

In data mining area, discovering frequent sequential patterns is a hot topic. In this problem, we find patterns which satisfy the minimum support in a list of transactions. A transaction consists of a series of data-sequences, where each data-sequences contains a set of items. A sequential pattern is a subset of data-sequences.
Ramakrishnan and Rakesh proposed a sequence mining algorithm that is faster than previous algorithms. The GSP(Generalized Sequential Patterns)[1] algorithm finds all sequential patterns that have a user-specified minimum support. Besides, they declare that GSP is much faster than the AprioriAll algorithm.
Though the later algorithms like Spade[2] and Prefixspan[3] has achieved better performance than GSP, we discover that there's lot to improve from the original algorithm. We present a strategy called *state pruning* in hash tree to eliminate extra comparison. Furthermore, we adjust parameters before building hash-tree in each iteration, which reduces a great amount of search space. Since we almost reinvent the original GSP algorithm, we call our algorithm $Hash^2$ GSP, which means using the quality of the hash-tree to save time. In our experiment, the $Hash^2$ GSP runs faster than Prefixspan algorithm.

2 Problems Statement

2.1 definitions

An itemset is a non-empty set of items. A sequence is an ordered list of itemsets. We denote a sequence s by $\langle s_1 s_2 s_3 s_n \rangle$, where s_j is an itemset. We also call s_j an element of the sequence. We denote an element of a sequence by $(x_1; x_2; :::; x_m)$, where x_j is an item. An item can occur only once in an element of a sequence, but can occur multiple times in different elements. An itemset is considered to be a sequence with a single element. We assume without loss of generality that items in an element of a sequence are in lexicographic order.
A sequence $\langle a_1 a_2 ::: a_n \rangle$ is a subsequence of another sequence $\langle b_1 b_2 ::: b_m \rangle$ if

there exists integers $i_1 < i_2 <:::< i_n$ in such that $a_1 \subseteq b_{i1}$, $a_2 \subseteq b_{i2}, ..., a_n \subseteq b_{in}$. For example, the sequence $\langle(3)(45)(8)\rangle$ is a subsequence of $\langle(7)(3,8)(9)(4,5,6)(8)\rangle$, since $(3) \subseteq (3,8)$, $(4,5) \subseteq (4,5,6)$ and $(8) \subseteq (8)$. However, the sequence $\langle(3)(5)\rangle$ is not a subsequence of $\langle(3,5)\rangle$ (and vice versa).

2.2 Input

We are given a database D of sequences called data-sequences. Each data-sequence is a list of transactions, ordered by increasing transaction-time. A transaction has the following fields: sequence-id, transaction-id, transaction-time, and the items present in the transaction.
For simplicity, we assume that no data-sequence has more than one transaction with the same transaction-time, and use the transaction-time as the transaction-identifier. We do not consider quantities of items in a transaction.

2.3 Sliding Windows

The sliding window relaxes the definition of a data-sequence contributes to the support of a sequence. Formally, a data-sequence $d = \langle d_1 \dots d_m\rangle$ contains a sequence $s = \langle s_1 \dots s_n\rangle$ if there exist integers $l_1 \leq u_1 < l_2 \leq u_2 < \dots < l_n \leq u_n$ such that

1.s_i is contained in $\bigcup_{k=l_i}^{u_i} d_k$, $1 \leq i \leq n$, and

2.transaction-time(d_{u_i})-transaction-time(d_{l_i}) $\leq$ window-size, $1 \leq i \leq n$.

Though this term is defined in the original paper, we do not discuss the specific implementation of the attribute.

2.4 Time Constraints

Time constraints restrict the time gap between sets of transactions that contain consecutive elements of the sequence. Given user-specified window-size, max-gap and min-gap, a data-sequence $d = \langle d_1 \dots d_m\rangle$ contains a sequence $s = \langle s_1 \dots s_n\rangle$ if there exist integers $l_1 \leq u_1 < l_2 \leq u_2 < \dots < l_n \leq u_n$ such that

1.s_i is contained in $\bigcup_{k=l_i}^{u_i} d_k$, $1 \leq i \leq n$,

2.transaction-time(d_{u_i})-transaction-time(d_{l_i}) $\leq$ window-size, $1 \leq i \leq n$,

3.transaction-time(d_{l_i})-transaction-time($d_{u_{i-1}}$) $>$ min-gap, $2 \leq i \leq n$, and

4.transaction-time(d_{u_i})-transaction-time($d_{l_{i-1}}$) $\leq$ max-gap, $2 \leq i \leq n$

2.5 Problem Definition

Given a database D of data-sequences, user-specified min-gap and max-gap time constraints, and a user-specified sliding-window size, the problem of mining sequential patterns is to find all sequences whose support are greater than the user-specified minimum support. Each such sequence represents a *sequential pattern*, also called a *frequent sequence*.
Given a frequent sequences $s = \langle s_1 \dots s_n\rangle$, it is often useful to know the "support relationship" between the elements of the sequence. That is, what fraction of

the data-sequences that support $\langle s_1 \dots s_i \rangle$ support the entire sequence s. Since $\langle s_1 \dots s_i \rangle$ must also be a frequent sequence, this relationship can easily be computed.

3 Algorithm GSP

The main structure of GSP is similar to AprioriAll. In the k-th iteration, the algorithm generates candidates with size of k and then scan the database to count their support. In AprioriAll, sequential patterns are generated by adding a frequent itemset each iteration. But GSP adds only one item to verified sequential patterns.
The algorithm makes multiple passes over the data. The first pass determines the support of each item. At the end of the first pass, the algorithm finds all the frequent items.
In the following pass, the algorithm begins with a seed set of frequent sequences found in the last pass. Then the algorithm generates potential sequences from these frequent sequences. We call these potential sequences candidates. Each candidate sequence has one more item than the frequent sequence defined in the last pass. The algorithm scans the data again to determine the support of candidate sequences. At the end of the scanning, the algorithm selects the actually frequent sequences from candidates. The newly found frequent sequences become the seed set of the next iteration.
The algorithm terminates, when no frequent sequence was found at the end of a pass or no candidates are generated.
The key steps of each iteration are candidate generation and counting candidates.

3.1 Candidate generation

We refer to a sequence with k items as a k-sequence.
Let L_k denote the set of all frequent k-sequences, and C_k the set of candidate k-sequences.
Given L_{k-1}, the set of all frequent $(k-1)$-sequences, we want to generate a superset of the set of all frequent k-sequences.
Candidates are generated in two steps:

1.**Join phase:** We generate candidate sequences by joining L_{k-1} with L_{k-1}. A sequence s_1 joins with s_2 if the subsequence obtained by dropping the first item of s_1 is the same as the subsequence obtained by dropping the last item of s_2. The candidate sequence generated by joining s_1 with s_2 is the sequence s_1 extended with the last item in s_2. The added item becomes a separate element if it was a separate element in s_2, and part of the last element of s_1 otherwise.

2.**Prune phase:** Given the set of candidate sequences, we will check whether their $(k-1)$-subsequence are frequent sequences. If there is no max-gap constrains, we also delete candidate sequences that have any subsequence without minimum support.

3.2 Counting Candidates

We read one data-sequence at a time and increase the support count of candidates contained in the data-sequence. We use two techniques to solve this problem.

1. Reducing the number of candidates in C by a hash-tree data structure.

2. We find whether a special candidates is a subsequence of d efficiently by transforming the representing of the data-sequence d.

3.2.1 Reduce the number of candidates by hash-tree

A node of the hash-tree either contains a list of sequences(left node) or a hash table(interior node). In a interior node, each nonempty bucket of the hash table points to another node. The root node is defined to be at depth 1, and an interior node at depth p points to nodes at depth $p + 1$.

Adding candidate sequences to the hash-tree. We use top-down algorithm to add candidate sequences to the hash-tree. At an interior node at depth p, we decide which branches to follow by applying the hash function to the *p-th* item in sequence and it start from the root. When the number of sequences in a leaf node exceed a threshold the leaf node will be an interior node.

Finding the candidates contained in a data-sequence. We apply different algorithm to different type of nodes, by which we can find all candidates contained in the data-sequences d.

• *Interior node, if it is the root:* Apply the hash function to each item in d, and recursively apply this procedure to the node in corresponding bucket. Since this is the potential start of a candidate, we do not need to consider the max-gap or min-gap

• *Interior node, if it is not the root:* Let t be the transaction time of the node we reached by hashing on an item x. Apply the hash function to each item in d whose transaction time is in $[t - window - size, t + max(window - size, max - gap)]$, and recursively apply this produce to the node in corresponding bucket.

• *Leaf node:* For each sequence s in the leaf, we just add s to answer set if it is contained by d by the algorithm in 3.2.2.

3.2.2 Checking whether a data-sequence contains a specific sequence

Let d be a data-sequence and let s be a candidate sequence.

Contain test algorithm: The algorithm checks whether s is contained in d, which alternates between two phase, and starts in the forward phase from the first elements. The algorithm is repeated until all elements are found.

• *Forward phase:* If the end-time of the elements just found is smaller than max-gap plus the start time of the previous elements, then the algorithm just deal with the next elements in s. If not, the algorithm switch to the backward phase.

• *Backward phase:* Let s_i be the current element in s and let t denote the end-time(s_i). The algorithm backtrack and "pull up" the previous element by finding the first set of transaction containing s_{i-1} whose end time is greater than t-max-gap. Pull up the previous elements if it is necessary for keeping the constraint satisfied, which is repeated until either the constraint the max-gap between the element just pulled up and the previous elements is satisfied or the constraint can not be satisfied. If the constraint is satisfied, the algorithm switch to forward phase, and find the next element of s_i in s. If not, the algorithm terminate.

Finding a single element: The algorithm accelerates the rate of finding the first occurrence of an element in a data-sequence by transform the representation of d as follows.

Creating an array whose elements are the items in the sequence. For each element in array stores a list of transaction time of corresponding item.

To find the first occurrence of an item after time t, we need to travel the list of the item until finds an item whose transaction time is greater than t. To find the first occurrence of an element in a data-sequence, the algorithm makes one pass through the items in the element and find the first occurrence of each item after time t.

This solution is better than simple matching. Because in this time-list matching algorithm, the sequence has only to be accessed once, while the sequence has to be accessed many time in the simple algorithm. Details of the algorithm is presented in the next section.

4 Implementation and Improvement

4.1 Representation of hash-tree

At first we implemented the whole original GSP algorithm, including the hash-tree, whereas the performance is not as good as we expected. The number of candidates are not reduced by hash-tree when encountering a new sequence from database. We check each candidate with the sequence from the database and increase their support count when they are contained by the sequence. But the checking step may cost much time since the number of candidate is very large. For example, if the number of frequent item is $L_1 = 300$, then the number of 2-item candidates is $L_1 * (L_1 - 1)/2 + L_1 * L_1 = 134850$, which is a huge number. If we compare every candidate with the every sequence from database, the algorithm may run over 10 minutes just return a small number of frequent 2-item sequence.

The huge gap between the number of candidates and real frequent patterns motivates us to improve the algorithm. The key step is reducing the number of comparison for each sequence from database. Since each sequence might match only a small number of candidate patterns, we needn't check every candidates with the sequence. A proper solution is first filtering the candidate patterns that mustn't be supported by the given sequence and then checking the rest candidate patterns one by one.

We first apply the hash-tree method proposed by the author[]. The data structure of the hash-tree is defined as

```
struct tree_node {
        std::vector<Pattern *> node_patterns;
        std::vector<struct tree_node *> children;
}
```

Each interior tree node contains a list of pointer pointing to their children. Leaf nodes do not have children but contain a list of candidate patterns. For a new coming sequence, we use the method mentioned above to search the hash-tree from the tree root.

Algorithm 1 Search along hash-tree

```
Require: Sequence $s$, hash-tree node $v$ and last hashed $idx$
 1: function SearchNode($v, s, idx$)
 2:     if $v$ is leaf node And $v$ has not been visited then
 3:         CheckPatterns($v.patterns$);
 4:     else
 5:         for $i = idx + 1 \to s.length$ do
 6:             if $idx \geq 0$ And $s[i].time - s[idx].time <= max\_gap$ then
 7:                 $child\_idx = $ Hash$((s[i]))$;
 8:                 $child = v.children[child\_idx]$;
 9:                 SearchNode($child, s, i$);
10:             end if
11:         end for
12:     end if
13: end function
```

We hash every remained items in the sequence, then move to the corresponding child and search with remained sequence. The searching process is recursive, thus for one node duplicate searching is possible. For instance, given a sequence $(1, 2)(3, 4)(5)$ and hash-tree with hash function $item\%3$, part of the hash searching process are drawn below:

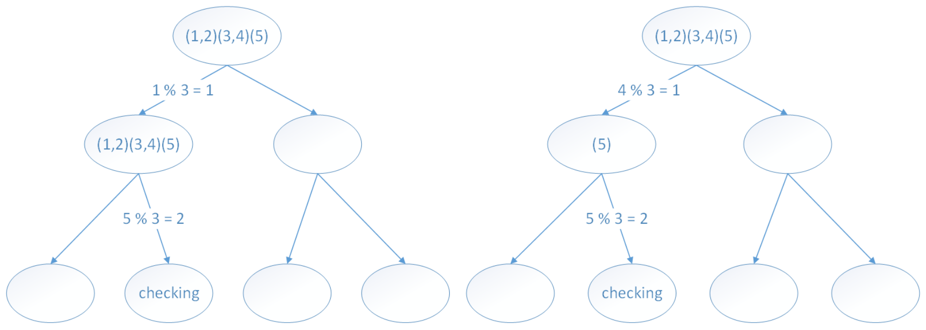


Figure 1: Hash searching

4.2 Reducing duplicate searching

From the figure we can find that though sequence $(1,2)(3,4)(5)$ was hashed in different ways, the final leaf node to be checked is the same. Indeed, not only the leaf nodes, the interior nodes may encounter duplicate visit. The duplicate visit cost much time, since each visit is a recursive function. So we want to eliminate the duplicate visit of nodes. That means we will not visit one interior node twice with the same state, in the meanwhile visit every potential leaf nodes. Here we define the condition that we shall stop further visiting.

• *item_index* : *idx* For on node v in hash-tree, now the sequence's idx^{th} item is hashed and the algorithm try further to apply recursive searching function on v.

Since for node v, we do not care the hashed items before reaching v. We just care which item was hashed last time. If the last hashed items of two distinct visit were hashed into the same position of the sequence, the following process of the searching is the same too. So for each sequence, we save all the different visiting record of each node when the searching function visiting them. Before searching along this node, we check whether this node has been visited with current sate. If so, we will stop this searching function. In order to record the visiting state of each node, we design the data structure as below.

```
struct tree_node {
        std::vector<Pattern *> node_patterns;
        std::vector<struct tree_node *> children;
        std::vector<bool> visited;
}
```

Vector *visited* is used to record all the *item_index* with which the searching function visit node v.

The pseudocode of searching with record state are followed as:

Algorithm 2 Search along hash-tree

```
Require: Sequence $s$, hash-tree node $v$ and last hashed $idx$
  function SearchWithSate($v, s, idx$)
      if $v$ is leaf node then
          CheckPatterns($v.patterns$);
      else
          for $i = idx + 1 \rightarrow s.length$ do
              if $idx \geq 0$ And $s[i].time - s[idx].time <= max\_gap$ then
                  $child\_idx = $ Hash($(s[i])$);
                  $child = v.children[child\_idx]$;
                  if $child.visted[i] = false$ then
                      $child.visted[i] = true$;
                      SearchNode($child, s, i$);
                  end if
              end if
          end for
      end if
  end function
```

After adding vector *visited* for every node, the recursive function will not visit one node with the same *item_index* twice. Take the search case below for example,

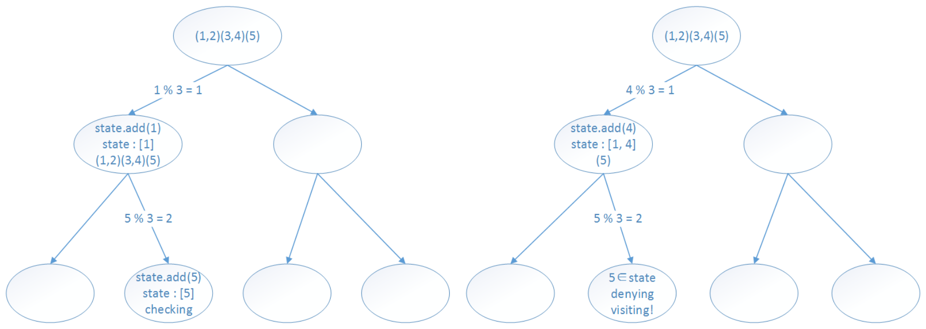


Figure 2: Hash searching with state

The figure in the left present one of the execution of recursive searching function. Two nodes add new state in this execution, including one leaf node. When another searching process try to visit the leaf node with the same state, it is denied. With the state vector of every node, our algorithm is much faster than the original one.
The following pseudo-code illustrates the procedure of checking a specific candidate in a sequence.

Algorithm 3 Check with time-list

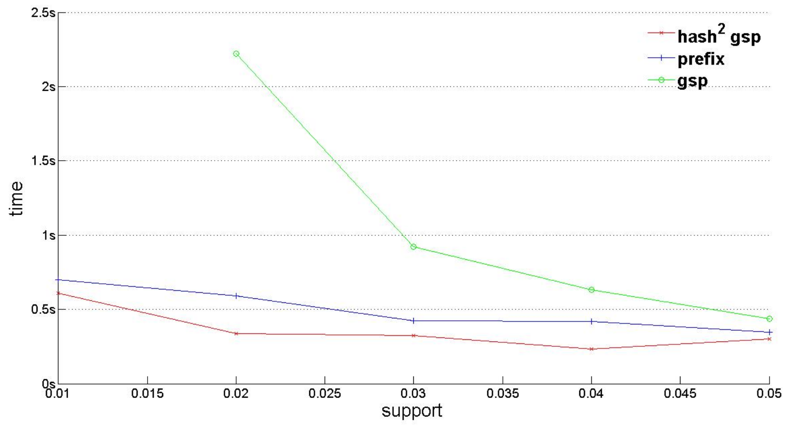
```
Require: Candidate $c$, Sequence $s$
    function SearchWithSate($c, s$)
        $start\_idx = 0, pos[] = NULL$;
3:      while true do
            $item\_set = c[start\_idx]$;
            while $item\_set! = c.end()$ do
6:              Find a point where $items$ in $item_set$ has interval $\leq$ WindowsSize
            end while
            Increase $pos[]$ for each position in $item\_set$;
9:          if do not find any $item\_set$ then
                return $false$;
            end if
12:         if $pos[start\_idx].time$ match $min\_gap$ and $max\_gap$ then
                $++start\_idx$;
            else if $pos[start\_idx].time$ exceed $max\_gap$ then
15:             $--start\_idx$;
            end if
            if $start\_idx == c.size()$ then
18:             return $true$.
            end if
        end while
21: end function
```

In **Line 8**, we increase the position of matching index in the sequence. Since the indexed will never decrease, we only have to traverse the sequence once.

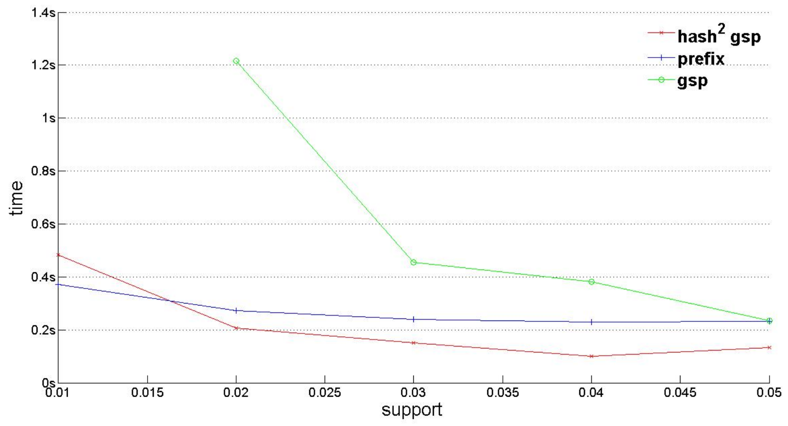
5 Experiments and Performance

Our experiments are on the environment of linux core 3.13.0-30-generic, with distribution of Ubuntu 14.04 LTS 64bit. The hardware platform is Intel Core i7-4600U CPU @ 2.10GHz.

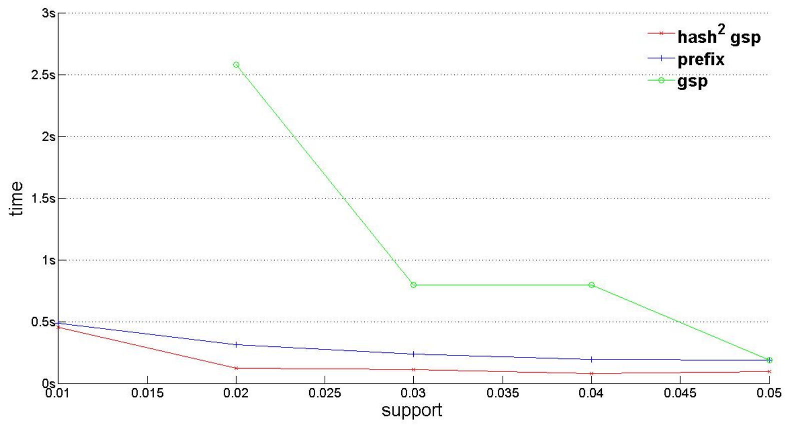
In our experiments, we compare three different algorithms by running data from [5], which are collected from real life. The traditional GSP algorithm and Prefixspan algorithm are implemented in the spmf [4]. Because the data are sparse, the support is set from 0.01 to 0.05. From the following results, we figure out that our $Hash^2$ GSP is very much faster than the original one. In the front of large data, the $Hash^2$ GSP provides a $5-10$ times speed-up against the original one. Furthermore, it also renders a better average performance than the Prefixspan algorithm.



(a) kosarak25k



(b) kosarak10k



(c) BMS1_spmf60k

The support vector is $(0.1, 0.2, 0.3, 0.4, 0.5)$ for $hash^2gsp$ and $prefix$, while the support vector of gsp is $(0.2, 0.3, 0.4, 0.5)$ because the time of support 0.1 of gsp is too large to be shown in these figures. In figure 3(a), the time vector of $hash^2gsp$ is $(0.299, 0.233, 0.323, 0.339, 0.611)$, the time vector of $prefix$ is $(0.345, 0.42, 0.425, 0.591, 0.7)$ and the time vector of gsp is $(0.436, 0.631, 0.921, 2.22)$. In figure 3(b), the time vector of $hash^2gsp$ is $(0.133, 0.099, 0.15, 0.207, 0.484)$, the time vector of $prefix$ is $(0.232, 0.23, 0.24, 0.272, 0.372)$ and the time vector of gsp is $(0.235, 0.382, 0.454, 1.217)$. In figure 3(c), the time vector of $hash^2gsp$ is $(0.093, 0.078, 0.109, 0.124, 0.453)$, the time vector of $prefix$ is $(0.188, 0.191, 0.235, 0.312, 0.485)$ and the time vector of gsp is $(0.188, 0.796, 0.797, 2.58)$.

6 Conclusion

In this project, we implement the GSP Algorithm (Generalized Sequential Pattern algorithm) and improve it.

The original version of the GSP[1] just reduced the number of checked candidates with hash-tree. We further study the mechanism of recursive searching and find that the execution of recursive function may visit one node for many times with the same state. This results duplicate search and the calling of recursive function costs much time.

We find the rule of calling recursive function and use vector in each hash-tree node to record the visit state. In this way, we sharply reduce the duplicate searching space. We compare the running time of our algorithm with several other algorithms and find that our algorithms is much faster than the original GSP. Moreover, our algorithm beats the prefix in the databases from [5]. Currently, our algorithm's efficiency heavily depends on the number of hashed branches and the max size that a hash-tree node can contain. We think there parameters are related with the distribution of items. So we may further study the relationship between parameters and the distribution of items. We expect this heuristics algorithm will performance better. If possible we will provide the upper bound of this heuristics algorithm based on the distribution of real data.

References

[1] Ramakrishnan Srikant, Rakesh Agrawal. "Mining Sequential Patterns: Generalizations and Performance Improvements." Advances in Database TechnologyEDBT'96, 1-17.

[2] Zaki, Mohammed J. "SPADE: An efficient algorithm for mining frequent sequences." Machine learning 42.1-2 (2001): 31-60.

[3] Pei, Jian, et al. "Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth." 2013 IEEE 29th International Conference on Data Engineering (ICDE). IEEE Computer Society, 2001.

[4] Philippe Fournier-Viger. ”SPMF: An Open-Source Data Mining Library.” www.philippe-fournier-viger.com/spmf/index.php?link=algorithms.php, spmf algorithm page, 2014.

[5] Philippe Fournier-Viger. ”SPMF: An Open-Source Data Mining Library.” http://www.philippe-fournier-viger.com/spmf/index.php, spmf homepage, 2014.