Mining Sequences of Item-sets

Juan Manuel Gimeno Illa[1] and Javier Béjar Alonso[2]

[1] Departament d'Informàtica i Enginyeria Industrial
Universitat de Lleida
jmgimeno@eup.udl.es

[2] Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
bejar@lsi.upc.es

Abstract. Given a sequence, mining sequential rules that account for that sequence has been a problem widely solved. Also many algorithms exist that, given a collection of item-sets, find the most frequent one. In this paper, we address the combined problem, that is, given a sequence of item-sets, find sequential rules that apply for that sequence. We do this by combining one existent algorithm for finding the most frequent item-sets with a sequential mining algorithm that takes into account substitutivity relationships between symbols in the alphabet.

1 Introduction

Mining frequent patterns in time series, sequences or transaction databases has been broadly studied in data mining research. The problems proposed in this discipline depend on the characteristics of the input sequence, the characteristics of the solution components and the definition of the associated frequency.

For instance, if the input sequence is a sequence over a given alphabet, we can define the solution components as subsequences of the input sequence and the frequency of a subsequence as the number of times this sequence appears in the given input sequence. Frequent mining in this context consists in finding all the subsequences whose frequency is greater than a minimum threshold [1, 6].

If the input sequence consists of a collection of sets of items, often called item-sets, we can define the solution components as item-sets whose frequency is the number of times each item-sets is included in the item-sets belonging to the collection. Frequent mining in this context consists in finding all the item-sets whose frequency is greater than a minimum threshold [2, 3].

In this paper, we propose a novel frequent mining algorithm that can be characterised as:

- the input sequence consisting is a *sequence* of *item-sets*
- the solution components are *subsequence of item-sets*
- the *frequency* of each subsequence is the number of times each subsequence *appears* in the sequence, where *appearance* means that the sets belonging to the subsequence are *included* in the sets belonging to the sequence

The rest of the paper is organised as follows: in section 2 we formally define the problem and outline the basic ideas behind our algorithm; sections 3 to 5 explain the different parts of our algorithm; section 6 solves an example step by step and, finally, section 7 gives conclusions and outlines some future work to do.

2 Problem definition

Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of all *items*. We define an *item-set* as a non-empty subset of I. A *sequence* is an ordered list of item-sets. A sequence S will be denoted by$< s_1, s_2, \ldots, s_L >$ where s_i is an item-set, *i.e.* $s_i \subseteq I$ for $1 \leq i \leq L$. L is the *length* of the sequence S. An *element* or *item-set* in the sequence s_i will be denoted as $\{x_1, x_2, \ldots, x_m\}$, where each x_j is an *item*. m is the *size* of the item-set s_i.

The consecutive item-sets $< s_i, \ldots, s_j >$ of $S =< s_1, s_2, \ldots, s_L >$ form a *subsequence* of S that starts at position i and ends at position j.

Given two sequences of item-sets R and S, with equal *length* L, we say R is *covered* [3] by S (or S is covers R) $\forall i\ r_i \subseteq s_i$ and we denote this by $R \sqsubseteq S$. The intuitive idea is that the item-sets in R are obtained by picking elements from the corresponding item-sets in S.

For instance, $< \{a, b\}, \{c\}, \{e, f\} >$ is covered by $< \{a, b, c\}, \{c, d\}, \{e, f\} >$ because $\{a, b\} \subseteq \{a.b.c\}$, $\{c\} \subseteq \{c, d\}$ and $\{e, f\} \subseteq \{e, f\}$; but it is not covered by $< \{a, u\}, \{c\}, \{e, f\} >$ because $\{a, u\} \not\subseteq \{a, b, c\}$.

We also define the *support* of a sequence R in S as the number of subsequences of S that cover R.

The support of the sequence $R =< \{a\}, \{b\} >$ in $S =< \{a, c\}, \{a, b\}, \{b, c\} >$ is 2, because R is covers the subsequences of S $< \{a, c\}, \{a, b\} >$ and $< \{a, b\}, \{b, c\} >$.

The problem to solve can be stated as: given a sequence of item-sets S and a *minimum support threshold minsup,* find all the sequences whose support in S is greater than or equal to *minsup.*

For instance, given the sequence S defined above and a *minsup* $= 2$, the solution would be the sequences: $< \{a\} >$,$< \{b\} >$, $< \{c\{>$ and $< \{a\}, \{b\} >$.

If we analyse the solution of the last example, we observe that both $< \{a\} >$ and $< \{b\} >$are subsequences of $< \{a\}, \{b\} >$. This is not a coincidence but a consequence of the *anti-monotone Apriori heuristic* [1]: *if any length k pattern is not frequent, any length $k + 1$ pattern containing it cannot be frequent.* That is, any subsequence of a frequent sequence must also be frequent.

In our problem, as the elements both in the input sequence and in the solution components are *item-sets,* we can reformulate this property in the following terms: *any item-set in any of the sequences of the solution must be frequent in the input sequence.*

Our algorithm uses this property in the following way: first, we find all the frequent item-sets in the whole sequence and next we use them as building blocks in order to find the frequent subsequences.

[3] Covering is the generalisation of set extension over sequences of sets.

3 Finding the frequent item-sets

Let us first characterise the subproblem to solve:

- the input is a *collection*[4] *of item-sets*
- the solution components are *item-sets*
- the frequency of an item-set is the number of item-sets in the collection that *include* it

So the subproblem consists in finding all the frequent item-sets in a collection of item-sets given a minimum threshold *minsup*. There are many algorithms that solve this problem [2, 3] so we will not further detail this part of the algorithm.

For instance, given the item-set collection
$[\{a, b\}, \{x, u\}, \{l, a, b, m, n\}, \{a, c\}, \{e, d\}, \{x, v\}, \{o, b, a, p, q\}, \{x, b\}, \{x, w\}]$
and a *minsup* threshold of 2, the *frequent item-sets* are $\{a\}, \{b\}, \{x\}$ with frequency 4 and $\{a, b\}$ with frequency 3.

At this point, if the whole problem was mining the collection above as a sequence, we would know that the solution of the whole problem would be sequences containing only these four item-sets. If only we could use these item-sets as *labels* to recode the input sequence, any algorithm for finding frequent subsequences over a finite alphabet would solve this subproblem. This idea can be applied, but not directly and it will be developed in the next section.

4 Recoding the sequence

First we will assign an unique *label* to each of the frequent sets found in the previous step. Let $F = \{f_1, f_2, \ldots, f_s\}$ the frequent item-sets, we will denote as l_{f_i} the *label* assigned to item-set f_i, and $L_F = \{l_{f_1}, l_{f_2}, \ldots, l_{f_s}\}$ will be the *label alphabet*. In our example $L_F = \{l_{\{a\}}, l_{\{b\}}, l_{\{x\}}, l_{\{a,b\}}\}$.

Next, we will assign a label belonging to the alphabet to each of the item-sets in the input sequence without losing or adding information about the frequent sequences of item-sets this input sequence contains.

Let us consider item-set $\{a, b\}$. Which label will be assigned to it? As it corresponds to a frequent item-set it is natural to use the same label, that is $l_{\{a,b\}}$. So the recoded input sequence will be $S' =< l_{\{a,b\}}, \ldots >$. And let us consider that, when using a frequent mining algorithm for sequences over finite alphabets, we are considering the frequency of sequence $< l_{\{a\}} >$. As $l_{\{a\}} \neq l_{\{a,b\}}$ the first appearance of $l_{\{a,b\}}$ in S' does not count as an appearance of $l_{\{a\}}$ but it might because $\{a\} \subseteq \{a, b\}$ and both $< \{a\} >$ and $< \{a, b\} >$are covered by $< \{a, b\} >$.

[4] In the whole problem we must take into account the ordering among the different item-sets in the sequence. Now this ordering is not important and we substitute the notion of sequence by the notion of collection or multi-set. The elements in a collection will be enclosed between [and].

4.1 Partial order

As the last example shows, the labels in the alphabet are not *independent* but *related* and, formally, they are related by a *partial order relationship.*

Let $L_F = \{l_{f_1}, l_{f_2}, \ldots, l_{f_s}\}$ be the *label alphabet*, we can define over the elements of L_F the following relationship:

$$l_{f_i} \preceq l_{f_j} \ \Leftrightarrow\ f_j \subseteq f_i$$

and $(L_F, \preceq)$ is a partial order because it is *reflexive*, *antisimetric* and *transitive.*

Over a partially ordered set we can define an element $m \in L_F$ to be *minimal* in L_F if :

$$\forall l \in L_F \ l \neq m \ \Rightarrow \ \neg(l \preceq m).$$

Intuitively, this partial order represents the relationship of *substitutivity*: if label $l_1 \preceq l_2$, l_1 can be substituted by l_2 when counting the number of appearances of l_2, because each appearance of l_1 is also an appearance of l_2. So a requirement of the mining algorithm will be that, when counting the support of a subsequence, consider all the labels it can substitute.

In the example above, the partial order consists in the following relationships: $l_{\{a,b\}} \preceq l_{\{a\}}$ and $l_{\{a,b\}} \preceq l_{\{b\}}$ and the unique minimal item-set is $l_{\{a,b\}}$.

4.2 Recoding

Given the *label alphabet* and the *substitutability relationship*, we proceed to recode the input sequence. For each item-set s_i in the sequence, we consider the set of labels *contained*[5] in it, that is, $L(s_i) = \{l_f \in L_F \mid f \subseteq s_i\}$ and then we purge all the labels in the set that cannot be substituted by any other label also in the set, that is, labels that are not minimal $L'(s_i) = \{l \in L(s_i) \mid l\, is\, minimal\, in\, L(s_i)\}$ The label to assign depends on the cardinality of this set.

$|L'(s_i)| = 0$ In this case, we assign the item-set the label # and no subsequence of the coded sequence containing # will ever be considered frequent. The reason to do so is the *Apriori* property. For instance, the fifth item-set in the sequence $s_5 = \{e, d\}$ contains no label of the alphabet, and will be recoded as #.

$|L'(s_i)| = 1$ The code assigned to the item-set is the label corresponding to this item-set. For instance, the third item-set in the sequence is $s_3 = \{l, a, b, m, n\}$ which contains the labels $L(s_3) = \{l_{\{a\}}.l_{\{b\}}, l_{\{a,b\}}\}$ and, after purging we get $L'(s_3) = \{l_{\{a,b\}}\}$ so we assign label $l_{\{a,b\}}$ to s_3.

$|L'(s_i)| > 1$ In this case, none of the labels can represent the same information as s_i. We must generate a new label for this item-set. We do so by assigning a new label to this item-set and, for all the labels in $L'(s_i)$, stating that those labels are greater than the current one. For instance, $s_8 = \{x, b\}$ and so we create a new label $l_{\{x,b\}}$ and we add the following *order* relationships: $l_{\{x,b\}} \preceq l_{\{x\}}$ and $l_{\{x,b\}} \preceq l_{\{b\}}$.

[5] Those are the labels corresponding to item-sets that are contained in it. I have abused the language so as not to repeat the word "item-set"many times.

After doing this, we have a *base alphabet* consisting of the labels[6] $l_{\{a\}}$, $l_{\{b\}}$, $l_{\{a,b\}}$, $l_{\{x\}}$and $l_{\{x,b\}}$; the *order relationships* $l_{\{a,b\}} \preceq l_{\{a\}}$, $l_{\{a,b\}} \preceq l_{\{b\}}$, $l_{\{x,b\}} \preceq l_{\{x\}}$ and $l_{\{x,b\}} \preceq l_{\{b\}}$; and the recoded sequence

$$S' =< l_{\{a,b\}}, l_{\{x\}}, l_{\{a,b\}}, l_{\{a\}}, \#, l_{\{x\}}, l_{\{a,b\}}, l_{\{x,b\}}, l_{\{x\}} >.$$

Now, we must find all the frequent sequential patterns in S' over the given alphabet and we must take the *partial order* into account when computing the support of the subsequences of S'. This is covered in the next section.

5 Finding the frequent subsequences

In order to mine the sequence we will modify a previous algorithm defined by Jaak Vilo [6] to take into account the *partial order* defined over the alphabet.

The *naïve* algorithm for finding all the frequent subsequences of a sequence would generate all the subsequences of the given sequence, compute the support of each one and output only those with support greater than *minsup*. This solution would be impractical for medium size sequences.

In order to do this generation and counting efficiently there exists a data structure that allows us to represent and manipulate all the subsequences of a given sequence in a very efficient way. This structure is called *suffix-tree.*

5.1 Suffix trees

A *suffix-tree* T for a character string[7] $S\$$ of length $l+1$ is a rooted directed tree with exactly l leaves numbered 1 to $l+1$. Each internal node, other than the root, has at least two children and each edge is labelled with a non-empty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix-tree is that, for any leaf i, the concatenation of the edge labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i. That is, spells out $< s_i, \ldots, s_{l+1} >$. It is possible to construct, for a given input sequence, the suffix-tree associated to it in *linear* time [4, 5].

So every prefix of a path from the root to a leaf spells out a different subsequence of the given sequence.

5.2 Algorithm for simple patterns

In [6], Jaak Vilo presents different algorithms dealing with finding frequent patterns in *nucleotide sequences.* In order to present the basic concepts used by his algorithm, and ours, we will first review his algorithm for finding simple patterns (*i.e.* patterns consisting only of simple characters). Both algorithms construct

[6] As # is only used to mark *non-frequent* patterns, this symbol is not considered to belong to the label alphabet.

[7] The last character \$ is added to make all suffixes of the sequence non nested, that is, that no suffix of the sequence is prefix of another suffix. From now on we will assume \$ exists in the sequence.

a *quadratic* version of the suffix-tree, called *suffix-trie*, in which each edge label has unit length, that is, it is a character.

Suffix trie Each node N in the tree can be identified with a string α over the trie label alphabet that spells out the labels on the path from the *root* to node N. We denote that node N by $N(\alpha)$. Hence $N(\alpha c)$ is the child of node $N(\alpha)$ that corresponds to extending the string α with character c. The *root* node represents the empty string λ.

Every node in the trie has some internal structure for representing the additional information about the relation it has with other nodes in the trie, and the substring corresponding to the node. We will use *dot-notation* to represent subfields, that is $N.parent$, $N.child$, $N.char$ and $N.sibling$. The substring α is spelled out by the character labels $N.char$ along the path from the root to the node $N(\alpha)$. $N(\alpha X).char = X$ and $N(\alpha X).parent = N(\alpha)$. Given node N, we denote its children by $N.child(c)$ meaning the child P of node N such that $P.char = c$. The *siblings* of node N can be identified by $N.sibling(c)$ where this is shorthand for $N.parent.child(c)$.

Also, each node has information about the occurrences of the substring it represents. Actually, each node $N(\alpha)$ has the list of occurrences that substring α has in the string. To represent the occurrence that ends at position j of string S we store number $j+1$ (that is, we store the position in which this substring can be extended). This list is denoted by $N.pos$.

The idea is that each node in the tree represents a different subsequence of the original subsequence. Two nodes in the tree corresponding to two subsequences.

Algorithm 1 Frequent substring generation for simple substrings

```
root = new node;
root.char = $\lambda$
root.pos = (1,2,...,$|S|$)
enqueue(Q,root)
while N = dequeue(Q)
  for-each character $c$
    set(c) = $\emptyset$
  for-each $p \in N.pos$
    add $p+1$ to set($S[p]$)
  for-each character $c$ such that $|set(c)| \geq K$
    N.child(c) = new node P with label P.char=c
    P.pos = set(c)
    enqueue(Q,P)
  delete N.pos
return root
```

Basic algorithm The algorithm constructs incrementally the suffix trie corresponding to the given sequence. Initially we only have the *root* node (that represents the empty suffix λ) that can be extended at every position in the sequence.

Then it groups these positions when expanding a node depending on the character present at the extended position and creates nodes for all of them (these nodes represent the subsequences found so far). Due to that the list of extension positions also represents the different appearances of the subsequence, those nodes whose list has length below the *minsup* threshold need not be further expanded.

So only the *partial suffix-trie* for the subsequences that pass the threshold is built. Pseudo-code is shown in algorithm 1.

5.3 Allowing for substitutions

Our algorithm is a generalisation of Vilo's algorithm that allows *group-characters* in the patterns. For instance, pattern `A[GC]T` *matches* both `AGT` and `ACT` because at the second position both `G` and `C` are allowed. Our generalisation consist in defining a *substitution relationship* (represented by the *partial order)* between characters[8]. For instance, pattern `AXT` covers the same patterns as the example given before if character `X` can be substituted by `G` or `C`.

Vilo's algorithm only allows the use of *group-characters* in the patterns but not in the input sequence. As our sequence is constructed by any symbol in the *alphabet* we must also consider this possibility. So our algorithm extends Vilo's in these two directions:

1. We allow a *substitutability* relationship among characters. Its only requirement is to be a *partial order.*
2. We allow characters in any level of the *partial order* both in the input sequence and in the patterns explored.

Computation of the extension positions In order to allow substitutions, we have to consider how the *extension positions* are calculated because a *non-minimal* character in the input sequence extends patterns formed by this character but also those formed by any minimal character that can substitute it. Only the minimals are considered because, if not, we could possibly count more than once a symbol in the input.

For instance, if symbol $l_{\{a\}}$ appears in the input at position *pos*, we must include *pos* not only in $set(l_{\{a\}})$ but also in $set(l_{\{a,b\}})$ due to the fact that we have $l_{\{a,b\}} \preceq l_{\{a\}}$ and $l_{\{a,b\}}$ is a *minimal* symbol.

So in the algorithm we consider first the case where the extension symbol is a *minimal symbol*, that is, a symbol that is not a substitutive for any symbol in the alphabet. For these symbols, we proceed as in the basic procedure (algorithm 2).

[8] Vilo's simple characters are equivalent to *minimal* characters in our formulation. Vilo's group-characters are equivalent to our *non-minimal* ones. But our *substitutivity* relationships allows for group characters inside groups characters not allowed in Vilo's formulation.

Algorithm 2 processing minimal_symbols

```
for-each minimal_symbol $m$ such that $|set(m)| \geq K$
  N.child(m) = new node P with label P.char=m
  P.pos = set(m)
  enqueue(Q,P)
```

Then we proceed with the *non minimal symbols.* These symbols get position appearances from themselves and (as they can appear in the input) and from the *minimal symbols* they can substitute. But this produces *over generalisation*: it constructs more patterns than necessary because, for any pattern, it creates all the patters obtained by substituting a character in the original pattern with a character that can substitute it.

For instance, given the sequence $< l_{\{a,b\}} >$ we obtain patterns $< l_{\{a,b\}} >$, $< l_{\{a\}} >$and $< l_{\{b\}} >$ all of them with *support* $= 1$. As we know the *substitutability* relationship only the first one is *informative* because the last two can be deduced from it.

The solution to this problem is to consider only patterns *generalising* other patterns when the generalised pattern has *more support* than the previous one. So we purge those whose support is not greater than the support of the concrete one (the purging procedure is shown in algorithm 3). This purging only affects the current symbol considered but not any symbol previously found.

Algorithm 3 processing non-minimal symbols

```
for-each non_minimal_symbol $s$
  positions = set($s$)
  max = 0
  for-each minimal_symbol $m$ such that $m \preceq s$
    positions = merge(positions,set($m$))
    max = maximum (max, $|set(m)|$)
  if $|positions| > max$ and $|positions| \geq K$
    N.child($s$) = new node P with label P.char=$s$
    P.pos = positions
    enqueue(Q,P)
```

6 Applying the algorithm

Now we must apply the algorithm to the recoded sequence

$$S' =< l_{\{a,b\}}, l_{\{x\}}, l_{\{a,b\}}, l_{\{a\}}, \#, l_{\{x\}}, l_{\{a,b\}}, l_{\{x,b\}}, l_{\{x\}}, \$ >$$

over the alphabet $L_F = \{l_{\{a\}}, l_{\{b\}}, l_{\{a,b\}}, l_{\{x\}}, l_{\{x,b\}}\}$, given the order relationships $l_{\{a,b\}} \preceq l_{\{a\}}$, $l_{\{a,b\}} \preceq l_{\{b\}}$, $l_{\{x,b\}} \preceq l_{\{x\}}$ and $l_{\{x,b\}} \preceq l_{\{b\}}$ whose *minimal* symbols are $l_{\{a,b\}}$ and $l_{\{x,b\}}$.

Initially we only have the *root* node with $root.char = \lambda$ and $root.pos = \{1, 2, \ldots 10\}$, as we have appended symbol \$ in the sequence. This is the only node we can expand, so we take it from the queue and compute the positions at which we can expand using the different characters. Scanning the input we get:

$$set(l_{\{a,b\}}) = \{2,4,8\}\ set(l_{\{x\}}) = \{3,7,10\}\ set(l_{\{a\}}) = \{5\}$$
$$set(l_{\{x,b\}}) = \{9\} \qquad set(l_{\{b\}}) = \emptyset$$

We can process the *minimal* characters now, and we construct node $N(l_{\{a,b\}})$ (a node for $l_{\{x,b\}}$ is not created because its position list has less length than *minsup*) and we point $root.child(l_{\{a,b\}})$ to it.

We have to extend the position list of each *non-minimal symbol* with those of the corresponding substitutive *minimal symbols*. For instance, $set(l_{\{b\}})$ that initially was empty, get positions from both $l_{\{a,b\}}$and $l_{\{x,b\}}$. So the new position lists are

$$set(l_{\{x\}}) = \{3,7,9,10\}\ set(l_{\{a\}}) = \{2,4,5,8\}\ set(l_{\{b\}}) = \{2,4,8,9\}$$

All these symbols pass the *minsup* threshold and are *informative* so we create nodes for them. Next we open node $N(l_{\{a,b\}})$ and we can extend it at positions $\{2, 4, 8\}$. The initial values for the extension positions are

$$set(l_{\{a,b\}}) = \emptyset \quad set(l_{\{x\}}) = \{3\}\ set(l_{\{a\}}) = \{5\}$$
$$set(l_{\{x,b\}}) = \{9\} \quad set(l_{\{b\}}) = \emptyset$$

and none of the *minimal symbols* passes the threshold. For the *non-minimal* symbols, the only one which passes the threshold is $l_{\{x\}}$ with positions $set(l_{\{x\}}) = \{3, 9\}$ and which is also informative. So we create $N(l_{\{a,b\}}l_{\{x\}})$.

Next we expand $N(l_{\{x\}})$, extendible at $\{3, 7, 9, 10\}$ and calculate its possible extensions. We get

$$set(l_{\{a,b\}}) = \{4,8\}\ set(l_{\{x\}}) = \{10\}\ set(l_{\{a\}}) = \emptyset$$
$$set(l_{\{x,b\}}) = \emptyset \qquad set(l_{\{b\}}) = \emptyset$$

and *minimal symbol* $l_{\{a,b\}}$ passes the threshold, so $N(l_{\{x\}}l_{\{a,b\}})$ is created. For the *non-minimal* we compute their possible extensions getting

$$set(l_{\{x\}}) = \{10\}\ set(l_{\{a\}}) = \{4,8\}\ set(l_{\{b\}}) = \{4,8\}$$

and only $l_{\{a\}}$ and $l_{\{b\}}$ pass the threshold but none of them is *informative*. For example, $set(l_{\{a\}}) = set(l_{\{a,b\}})$, so we don't add any new node to the trie. Figure 1 shows the trie constructed in the example.

The patterns found by the algorithm are

$$< \{a,b\},\{x\} > < \{x\},\{a,b\} > < \{a\},\{x\} > < \{b\},\{x\} >$$

all with $support = 2$ but the last which has $support = 3$.

7 Conclusions and future work

In this paper we have presented an algorithm for finding frequent subsequences in sequences of item-sets. This algorithm consists in the combination of two algorithms for solving two different frequent mining problems:

- finding all the frequent item-sets in a collection of item-sets
- finding all the frequent subsequences over a character set with a substitutivity relationship

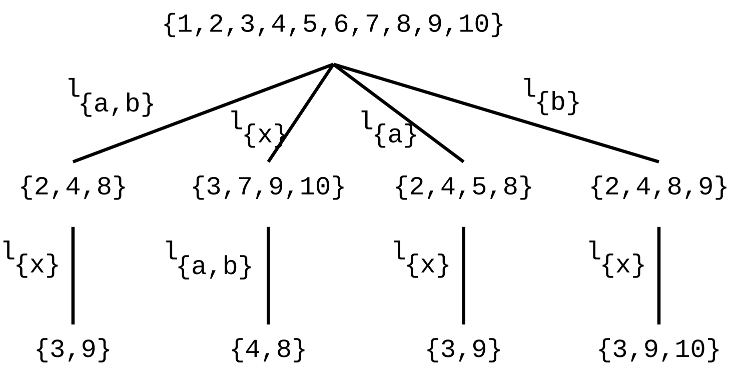

Fig. 1. Trie generated by the algorithm

The later algorithm is also a contribution of this paper.

Future work must be done to further improve the efficiency of the algorithm both in space and time. For instance in the example, we obtain patterns $< l_{\{a,b\}}, l_{\{x\}} >$ and $< l_{\{a\}}, l_{\{x\}} >$, both of them with the same support and, clearly, the later can be deduced from the former. Our *purging procedure* is not powerful enough to cope with this case. Better purging techniques must be developed. Parallelisation by clustering can also been considered.

References

1. R.Agrawal and R.Srikant. *Fast algorithms for mining sequential rules.* In VLDB'94, 487-499. 1994.
2. J. Han, J. Pei and Y.Yin. *Mining frequent patterns without candidate generation.* In 2000 ACM SIGMOD Intl. Conference on Management of Data, pp 1-12, 2000.
3. J.Pei, J.Han and R.Mao. *CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets.* ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discover, 21-30,2000.
4. E.M.McCreight. *A space-economical suffix tree construction algorithm.* Journal of the ACM, 23:262-272, 1976.
5. E.Ukkonen. *On-line construction of suffix-trees.* Algorithmica, 14:249-260, 1995.
6. J.Vilo. *Discovering Frequent Patterns from Strings.* Technical Report C-1998-9, Department of Computer Science, University of Helsinki, 1998.