Abstract - Traditional pattern growth-based approaches for sequential pattern mining derive length- \((k + 1)\) patterns based on the projected databases of length-\(k\) patterns recursively. At each level of recursion, they unidirectionally grow the length of detected patterns by one along the suffix of detected patterns, which needs \(k\) levels of recursion to find a length-\(k\) pattern. In this paper, a novel data structure, UpDown Directed Acyclic Graph (UDDAG), is invented for efficient sequential pattern mining. UDDAG allows bidirectional pattern growth along both ends of detected patterns. Thus, a length-\(k\) pattern can be detected in \(\lfloor \log_2 k + 1 \rfloor\) levels of recursion at best, which results in fewer levels of recursion and faster pattern growth. When \(\text{minSup} \) is large such that the average pattern length is close to 1, UDDAG and PrefixSpan have similar performance because the problem degrades into frequent item counting problem. However, UDDAG scales up much better. It often outperforms PrefixSpan by almost one order of magnitude in scalability tests. UDDAG is also considerably faster than Spade and LapinSpam. Except for extreme cases, UDDAG uses comparable memory to that of PrefixSpan and less memory than Spade and LapinSpam. Additionally, the special feature of UDDAG enables its extension toward applications involving searching in large spaces.

Indexterms : data mining algorithm, directed acyclic graph, performance analysis, sequential pattern, transaction database.

GJCST-C Classification : H.2.8
A Novel Approach for Scalability – Two Way Sequential Pattern Mining using UDDAG

Dr. P. Raguraman *, Mr. S. Hariharan § & Dr. J. Jaya A Celin ♦

Abstract - Traditional pattern growth-based approaches for sequential pattern mining derive length- \((k + 1)\) patterns based on the projected databases of length-\(k\) patterns recursively. At each level of recursion, they unidirectionally grow the length of detected patterns by one along the suffix of detected patterns, which needs \(k\) levels of recursion to find a length-\(k\) pattern. In this paper, a novel data structure, UpDown Directed Acyclic Graph (UDDAG), is invented for efficient sequential pattern mining. UDDAG allows bidirectional pattern growth along both ends of detected patterns. Thus, a length-\(k\) pattern can be detected in \(|\log_2 k + 1|\) levels of recursion at best, which results in fewer levels of recursion and faster pattern growth. When \(\text{minSup} \geq k\) is large such that the average pattern length is close to 1, UDDAG and PrefixSpan have similar performance because the problem degrades into frequent item counting problem. However, UDDAG scales up much better. It often outperforms PrefixSpan by almost one order of magnitude in scalability tests. UDDAG is also considerably faster than Spade and LapinSpam. Except for extreme cases, UDDAG uses comparable memory to that of PrefixSpan and less memory than Spade and LapinSpam. Additionally, the special feature of UDDAG enables its extension toward applications involving searching in large spaces.

Index terms: data mining algorithm, directed acyclic graph, performance analysis, sequential pattern, transaction database.

I. Introduction

Sequential pattern mining is an important data mining problem, which detects frequent subsequences in a sequence database. A major technique for sequential pattern mining is pattern growth. Traditional pattern growth-based approaches (e.g., PrefixSpan) derive length- \((k + 1)\) patterns based on the projected databases of a length-\(k\) pattern recursively. At each level of recursion, the length of detected patterns is grown by 1, and patterns are grown unidirectionally along the suffix direction. Consequently, we need \(k\) levels of recursion to mine a length-\(k\) pattern, which is expensive due to the large number of recursive database projections. In this paper, a new approach based on UpDown Directed Acyclic Graph (UDDAG) is proposed for fast pattern growth. UDDAG is a novel data structure, which supports bidirectional pattern growth from both ends of detected patterns. With UDDAG, at level \(i\) recursion, we may grow the length of patterns by \(2i\) at most. Thus, a length-\(k\) pattern can be detected in \(|\log_2 k + 1|\) levels of recursion at minimum, which results in better scale-up property for UDDAG compared to PrefixSpan. Our extensive experiments clearly demonstrated the strength of UDDAG with its bidirectional pattern growth strategy. When \(\text{minSup} \geq k\) is large such that the average length of patterns is very small (close to 1), UDDAG and PrefixSpan have similar performance because in this case, the problem degrades into a basic frequent item counting problem. However, UDDAG scales up much better compared to PrefixSpan. It often outperforms PrefixSpan by one order of magnitude in our scalability tests. UDDAG is also considerably faster than Spade and LapinSpam. Except for some extreme cases, the memory usage of UDDAG is comparable to that of PrefixSpan. UDDAG generally uses less memory than Spade and LapinSpam. UDDAG may be extended to other areas where efficient searching in large sequence spaces is necessary.

II. Related Work

The problem of sequential pattern mining was introduced by Agrawal and Srikant [1]. Among the many algorithms proposed to solve the problem, GSP [17] and PrefixSpan [13], [14] represent two major types of approaches: a priori-based and pattern growth-based. A priori principle states that any supersequence of a non-frequent sequence must not be frequent. A priori based approaches can be considered as breadth-first traversal algorithms because they construct all length-\(k\) patterns before constructing length- \((k+1)\) patterns. The AprioriAll algorithm [1] is one of the earliest a priori based approaches. It first finds all frequent item sets, transforms the database so that each transaction is replaced by all frequent item sets it contains, and then finds patterns. The GSP algorithm [16] is an improvement over AprioriAll. To reduce candidates, GSP only creates a new length- \(k\) candidate when there are two frequent length- \((k-1)\) sequences with the prefix of one equal to the suffix of the other. To test whether a candidate is a frequent length- \(k\) pattern, the support of each length- \(k\) candidate is counted by examining all the sequences. The PSP algorithm [12] is similar to GSP except that the placement of candidates is improved through a prefix tree arrangement to speed up pattern

Authors * : Assistant Professor, Department of Computer Science, Sri Sankara Arts and Science College, Enathur, Kanchipuram, TN, India. E-mail: contactragu84@gmail.com

Author ‡ : Associate Professor, Department of IT Noorul Islam University, India.

© 2013 Global Journals Inc. (US)
discovery. The SPIRIT algorithm [9] uses regular expressions as constraints and developed a family of algorithms for pattern mining under constraints based on a priori rule. The SPaRSε algorithm [3] improves GSP by using both candidate generation and projected databases to achieve higher efficiency for high pattern density conditions.

### III. Problem Definition

#### a) Updown Directed Acyclic Graph-Based Sequential Pattern Mining

UDDAG-based pattern mining approach, which first transforms a database based on frequent item sets, then partitions the problem, and finally, detects each subset using UDDAG. The absolute support for an item set in a sequence database is the number of tuples whose sequences contain the item set. An item set with a support larger than minSup is called a frequent item set (FI) set. Based on frequent item sets, we transform each item set in each sequence with the ids of all FIs contained in the item set.

Let {x_1, x_2, ..., x_t} be the frequent item sets in D, D' be its transformed database, substituting the ids of each item pattern contained in D' with the corresponding item sets, and denoting the resulted pattern set by P', we have P = P'.

Based on frequent item sets, we transform each sequence in a database D into an alternative representation. Steps involved in Database Transformation:

1. Find the set of frequent items in D.
2. Assign a unique id to each FI in D and then replace each item set in each sequence with the ids of all the FIs contained in the item set.

<table>
<thead>
<tr>
<th>Seq. Id</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;1 (1,2,3) 1,3 4 (3,6)&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;1 (1,4) 3 (2,3) 1,5&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;3 (5,6) 1 (2,7) 4 (6,3) 3 2&gt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;5 (7,1,6) 3 2 3&gt;</td>
</tr>
</tbody>
</table>

**Table 1**: Sequence Database

For the database in Table 1, the FIs are: (1),(2), (3), (4), (5), (6), (1,2), (2,3).

By assigning a unique id to each FI, e.g., (1)-1, (1,2)-2, (2)-3, (2,3)-4, (3)-5, (4)-6, (5)-7, (6)-8, we can transform the database as shown in Table 2 (infrequent items are eliminated).

#### i. Problem Partitioning

**Definition**

Let (x_1, x_2, ..., x_t) be the frequent item sets in a database D, x_1 < x_2 < ... < x_t, the complete set of patterns (P) in D can be divided into t disjoint subsets. The ith subset (denoted by P_i, 1 < = i < = t) is the set of patterns that contains x_i and FIs smaller than x_i.

#### ii. Definition (Projected database)

The collection of all the tuples whose sequences contain an item set x in a database D is called x-projected database, denoted by xD.

The total number of different Frequent Items FIs in the Sequential Database are found by Database Transformation module. For a database with n different frequent items, its patterns can be divided into n disjoint subsets. The ith subset (1 < i < n) is the set of patterns that contain i (the root item of the subset) and items smaller than i. Each subset i of the problem is mapped into a projected database denoted by iD.

P is partitioned into eight subsets as there are 8 FIs in Table 2, the one contains 1 (P1), the one contains 2 and smaller ids (P2), ..., and the one contains 8 and smaller ids (P8).

#### b) UpDown Directed Acyclic Graph

#### i. Definition

Directed acyclic graph (UDDAG) is a graphical approach that represents patterns as vertices and contain relationships as directed edges in between vertices.

Given an FI x and xD, an UpDown Directed Acyclic Graph based on P_x, denoted by x-UDDAG, is constructed as follows:

1. Each pattern in P_x corresponds to a vertex in x-UDDAG. <x> corresponds to the root vertex, denoted by V_x.
2. Let $P_u$ be the set of length-2 patterns ending with $x$ in $P_x$. Add a directed edge from $V_x$ to $U_v$ called up root child of $V_x$. This represents the common prefix set to root $X$.

3. Let $P_d$ be the set of length-2 patterns starting with $x$ in $P_x$, add a directed edge from $V_x$ to $V_d$ called a down root child of $V_x$. This represents the common suffix set to root $X$.

Such a DAG can be recursively constructed in an efficient way to derive the contain relationship of patterns.

**Example of UDDAG Construction**

By concatenating the patterns \{<3>,<4>,<5>,<4 5>\} of Pre($S_D$) with $8$, we derive patterns \{<3 8>, <4 8>, <5 8>, <4 5 8>\} in figure-a.

By concatenating patterns \{<3>, <4>, <6>, <3 6>, <4 6>\} of Suf($S_D$) with $8$, we derive patterns \{<8 3>, <8 4>, <8 6>, <8 3 6>, <8 4 3>, <8 4 6>\} in figure-b.

**Example: UDDAG based Pattern Mining**

In the $i$th subset, each pattern in the projected database ($^iD$) can be divided into two parts, prefix and suffix of $i$.

The collection of all the prefix/suffix tuples of a frequent item set $X$ in $^iD$ is called the prefix/suffix-projected database of $x$, denoted by Pre($^iD$) / Suf($^iD$).

To detect the sequential pattern in projected database ($^iD$) $P_x$,
- Detect patterns in Pre($^iD$) called pattern prefix (PP).
- Detect pattern in Suf($^iD$) called pattern suffix (PS)
- The above steps are repeated recursively until no frequent items are found in the pre($^iD$) / suf($^iD$).
- Combine the patterns of all the iterations to derive $P_x$.

The complete set of patterns is the union of patterns of the all subsets or projected database ($^iD$) detected above.

The Apriori property is used to reduce the number of candidate sets to be considered.

**Example for Pattern Mining**

Assuming $^iD$ is,
1. \(<4 5 8 3 6>\)
2. \(<3 4 5 8 3 1 5>\)
3. \(<3 8 2 4 6 3>\)

4. \(<2 8 4 3 6>\).

The prefix subsequences of 8 in $^iD$, or Pre($^iD$) is:
\{<3 8>, <4 8>, <5 8>, <4 5 8>\} the patterns with 8 at the end.

The suffix subsequences of 8 in $^iD$, or Suf($^iD$) is:
\{<8 3>, <8 4>, <8 6>, <8 3 6>, <8 4 3>, <8 4 6>\} the patterns with 8 at the beginning.

The patterns with 8 in between the beginning and end of each pattern is:
\{<3 8 3>, <4 8 3>, <5 8 3>, <4 5 8 3>\}.

**Algorithm 1: UDDAG based pattern Mining.**

**Input**: A database $D$ and the minimum support

**Output**: $P$, the complete set of patterns in $D$

**Method**: findP ($D$, minSup)

\[ P = \emptyset \]

FISet = $D$.getAllFI(minSup);
$D$.transform();
for each FI $x$ in FISet{
UDVertexrootVT = newUDVertex($x$);
findP($D$.getPreD($x$), rootVT, up, minSup);
findP($D$.getSufD($x$), rootVT, down, minSup);
findPUDDAG(rootVT);
P = P U rootVT.getAllPatterns();
}

**Subroutine** findP ($PD$, rootVT, type, minSup)

\[ FISet = PD$.getAllFI(minSup); \]

for each FI $x$ in FISet{
UDVertexcurVT = newUDVertex($x$);
findP($D$.getPreD($x$), curVT, up, minSup);
findP($D$.getSufD($x$), curVT, down, minSup);
findPUDDAG(curVT);
}

© 2013 Global Journals Inc. (US)
findPUDDAG(rootVT)
upQueue.enQueue (rootVT.upChildren)
while(!upQueue.isEmpty())
upVT=upQueue.deQueue()
if(upVT.upParent==rootVT)
downQueue.enQueue(rootVT.downChildren)
else if (upVT.downParent==null)
downQueue.enQueue(upVT.upParent.VDVS)
else downQueue.enQueue(upVT.upParent.VDVS  n
upVT.downParent.VDVS)
while(!downQueue.isEmpty())
downVT=downQueue.deQueue()
if(isValid(upVT, downVT)
curVT=new UDVertex (upVT, downVT)
upVT.addVDVS(downVT)
if(upVT.upParent==rootVT)
downQueue.enQueue(downVT.children)
if(upVT.VDVS.size>0)upQueue.enQueue(upVT.children)

IV. Performance Evaluation
We conducted an extensive set of experiments to compare our approach with other representative algorithms. All the experiments were performed on a Core 2 Duo 3.0 GHz Quad Core Intel Xeon Server and 16 GB memory. The algorithms we compared are PrefixSpan, Spade, and LapinSpam, which were all implemented in C++. Two versions of UDDAG were tested. Minor changes have been made to adapt Space to Windows.

V. Conclusion
In this paper, a novel data structure UDDAG is invented for efficient pattern mining. The new approach grows patterns from both ends (prefixes and suffixes) of detected patterns. This is different from traditional approaches that only grow patterns from one end. The data sets were generated by maintaining all except one of the parameters as shown in Table 4 fixed, and exploring different values for the remaining ones.

The performance evaluation showed that UDDAG is faster than PrefixSpan and Spade in most cases. The scalability of UDDAG was also tested, and it was found to be comparable to PrefixSpan.

One major feature of UDDAG is that it supports efficient pruning of invalid candidates, a promising approach for applications involving searching in large spaces. This has great potential to related areas of data mining and artificial intelligence. In the future, we expect to further improve UDDAG-based pattern mining algorithm as follows: 1) Currently, FDI detection is independent from pattern mining. Practically, the knowledge gained from FDI detection may be useful for pattern mining. In the future, we will integrate the solutions of the two so that they can benefit each other. 2) Different candidate verification strategies may have different impacts to the efficiency of the verification strategy. In the future, we will study more efficient verification strategies.