Frequent Itemset Mining in Tree-Like Sequences of Complex Objects

Tokpo Ewoenam Kwaku
ewoenamkwaku.tokpo@studenti.unitn.it
Department of Computer Science
University of Trento
196504

Abayneh Getnet Beyene
abaynehgetnet.beyene@studenti.unitn.it
Department of Computer Science
University of Trento
196813

ABSTRACT

This paper presents an algorithm for mining frequent itemsets in tree-like sequences of complex objects. Several algorithms have been proposed for mining frequent itemsets. However, in this project we seek to focus on data inputs that are in tree-like sequences and the goal is to identify frequent itemsets of sequential patterns that are present in the data. A typical data input of this form consists of a several trees; a forest, each tree representing a discrete transaction in which each node of the tree is a record containing a tuple of key value pairs. The algorithm proposed in this paper finds the frequent patterns in which the items in each record appear in relation to the appearances of other items in the same transaction in a sequential manner. We are interested in identifying the frequent chain of occurrences of items in the data.

Keywords

Tree mining, Frequent itemset mining, Sequential pattern mining

1. INTRODUCTION

With so much data around us, it has become increasingly relevant to make sense of data and to gather as much information as we can from data. Every piece of data tells its own story and one of the important things we can learn from data is that various patterns keep repeating and if we look carefully, we can find useful trends in data. These trends or patterns are useful because they allow us to anticipate certain occurrences and to make useful predictions for the future. These predictions can save time, cost and a lot of effort which will otherwise be spent without proper information. This has made mining of tree-like sequences a very interesting topic to delve into. Finding frequent patterns is very relevant in a number of applications. From advertisement, to healthcare, to sales, banking and other important aspects of life. Because a lot of daily transactions are also carried out in a sequential manner, there is a lot of information we can derive from finding patterns in such sequences. Again, data is also increasingly being transferred in tree-like formats; a typical example being XML. This makes mining of frequent sequences for tree-like sequences very vital in understanding data presented in such formats [3] . Some specific practical applications of Frequent itemset mining in tree-like sequences are: studying the spread of vector-borne diseases, weblog analysis, spatio-temporal data mining and phylogenetic tree mining [8].

Most existing studies have focused on mining of frequent itemsets in general or mining substructures in trees. However, the focus of this paper is on mining frequent items in tree-likesequences of complex objects. In this, the data is in the form of trees. Each tree is a transaction and every node of the tree represents a record which is basically an itemset. A record could be a set of values of the form $\{i_1, i_2, i_3, ..., i_n\}$ or a set of key-value pairs $\{k_1 : v_1, k_2 : v_2, k_3 : v_3, ..., k_n : v_n\}$. Each transaction is uniquely identified by a transaction id. Each record in a transaction is also uniquely identified by a record id. Given such data as our input, our interest is to find the frequent sequences of items and pattern in which they appear. This can be useful in mining association rules and making predictions on subsequent data inputs.

To achieve this, we have developed an algorithm called FSP-Scop. The FSP-Scop algorithm mines for frequent itemsets in tree-like sequences mainly by adopting two traditional data mining approaches. The first stage mines for frequent itemsets for every node. Items that are not frequent are thus pruned for the nodes. This gives a compact set with which every node can be identified. To do this we used the Fp-growth algorithm proposed in [2] which is a popular and one of the fastest itemset mining algorithm. The second stage involves using various tree mining approaches to extract frequent substructures or frequent subtrees in the data. With this we get frequent sequences of items in the data. The main tree mining techniques we used in this part include:

1. A tree reduction technique proposed in [3] to convert a tree into a sequence using the.
2. The Gap_BIDE algorithm for efficiently mining closed subsequences with gap constraints developed in [6] to mine the frequent sequence from the reduced tree.

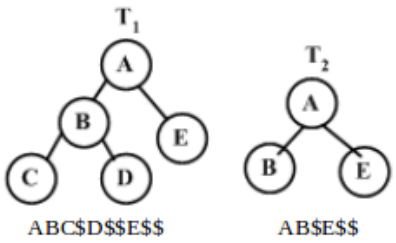

Figure 1: Pre-order encoding of 2 trees: T_1 and T_2.

The algorithm is explained with further details in section 3.

1.1 Preliminaries and Definitions

Trees: A tree is a directed acyclic connected graph $S = (V, E)$. V is a set of vertices or nodes and $E = (u, v)|u, v \in V$ is a set of edges. Each edge shows a relationship between two nodes. A distinguished node $r \in V$ is called the root and it has no incoming edge. There is a unique path that exists from a given node $x \in V$ to r. node u is an ancestor of node v if there is a path from u to v. v is called the decendant of u. u is the parent of v if $(u, v) \in E$, thus an edge exists between u and v. v is therefore a direct decendant of u and is called the child of u [8].

Itemset: An itemset is a subset of a set of items. Given a set ot items $I = \{i_1, i_2, i_3, ..., i_n\}$, an itemset P is a subset of I i.e. $P \subseteq I$. The number of items in an itemset is considered as the size of the itemset [8].

Sequence: A sequence is an ordered list of items denoted by $< s_1 s_2 ... s_n >$ where s_i is an itemset. Sequence $< a_1 a_2 ... a_n >$ is said to be the subsequence of sequence $< b_1 b_2 . b_n >$ if $\exists$ $i_1 < i_2 < ... < i_n$ such that $a_1 \subseteq b_{i_1}, b_{i_2}, ..., a_n \subseteq b_{i_n}$ [3].

Induced subtrees: A tree T_1 is an induced subtree of tree T if and only if T_1 preserves the parent-child relationship of T; such that for $v_1, v_2 \in V$, $preorder(v_1) < preorder(v_2)$ in T_1 if and only if $preorder(v_1) < preorder(v_2)$ in T [9]. The FSP-Scop algorithm proposed in this paper mines for induced subtree patterns. This is because induced subtrees give a lot more useful and interesting patterns than embedded subtrees [8].

Pre-order-string: The preorder string is an encoding form for rooted ordered trees [9]. It is based on the depth-first traversal approach. A special character known as the end flag is used to mark a step up from a node to its parent also known as a back-track. In this paper, we use the '$' sign as the end flag. An example of the pre-order encoding used in [9] is shown in figure 1.

Support:For a given itemset I, the support of I is the fraction of transactions in the dataset D that contains I with respect to the total number transactions in D expressed as a percentage . The support count of I is the number of transactions that contain I in D. The minimum support count is the lowest threshold support an item must have to be considered frequent. Formally, Let $I = \{i_1, i_2, i_3, ..., i_n\}$. Let $D = \{t_1, t_2, t_3, ..., t_n\}$. $k = |I|$. $supp(I) =\| \{t \in D|I \subseteq t\}m \| / \| \{t \in D\} \|$

2. RELATED WORKS

Related works under this topic are based on two main approaches; the Apriori approach [1] and the FP-growth approach [4]. The A prior approach involve two main strategies: the extention and the join strategies for candidate generation [5]. The join involves combining two frequent trees whilst the extension strategy uses tree extension or itemset extension to generate candidates [8]. The tree extension also involves two different ways. In the first way,a new node is added to the tree so that it becomes the sibling of the node in the rightmost part of the tree. This is called right path extension. In the other way, a new node is added as a child to the rightmost node in the tree. This is done in line with the candidate generation process in apriori algorithms. The itemset extension process involves adding a new item to the rightmost node's itemset [8].

The Fp-growth approach adopts a divide and conquer pattern growth principle. The Fp-growth approach avoids the candidate genration process of the apriori approach by making only two passes through the database thereby saving time and resources needed for candidate generation [8].

One of the most efficient algorithms developed in this field is the IMIT algorithm developed in [8] for frequent pattern mining in attributed trees. Here, the algorithm mines frequent patterns in attributed trees using a series of steps. In the first step, a set S containing all subtrees of size 1 is built by scanning the entire dataset D. It then loops to process every candidate in the set. The algorithm then carries out a frequency and canonical test where frequent items are added to a set L which is a set of solutions. The frequent candidates are extended using the different extension techniques explained above to generate new candidates in the subsequent iterations [8]. The IMIT algorithm is efficient in terms of enumerating all solutions. It however has a huge search space. Variations of this algorithm have been developed to address some of its limitations. Some Variationas include the IMIT-CLOSED and IMIT-CONTENT [8]. These variations have to compromise on efficiency to overcome the space constraint.

One other interesting algorithm to consider in this field is the FAG-gSpan [7].This algorithm was originally developed for graphs but shares very interesting resemblance with the topic. It has also been a building block for solving for tree-like sequences. The FAG-gSpan is the combination of two algorithms: gSpan for the enumeration of labelled patterns and the QFIMiner for generating quantitative itemsets to be assigned to the vertices. It utilizes rightmost extension to a pattern of labeled graphs p to enumerate patterns of labeled graphs p'. FAG-gSpan basically converts the list of vertices by replacing their attribute names with the order of the vertices in using depth-first traversal. It then generates a labeled graph by extracting the dense clusters and assigns the quantitative clusters to the labeled graphs. It uses the QFIMiner to obtain the quantitative itemsets in the vertex [7]. The obvious problem with this algorithm is the fact that it is better suited for graph problems. However the intuition behind the algorithm is worth taking not of.

2.1 Problem Statement

In this section, we are going to give a brief explanation of the problem and give a formal representation to it. Frequent itemset mining in tree-like sequences of complex objects is much related to mining of frequent itemset such as in market basket analysis in the sense that in both cases we deal with a set of transactions which are made up of items. In this case however, the problem is more complex than in the former due to the introduction of two key factors. The first is that the data is structured in a tree-like manner where each tree in the dataset represents a transaction. The second factor is that each transaction is made up of a set of records instead of items. These records contain attributes. The items are the attributes of the records which are key:value pairs or a set of items.

Now to define the problem formally: Given a set of transactions D such that $D = \{T_1, T_2, T_3, ..., T_n\}$, where T_i is a transaction in D. $T_i = (V_i, E_i)$ where V is the set of nodes in T and E is the set of edges which indicate the parent-child relationship between nodes. such that if u is a child of v, then $\exists(u, v) \in E$. $v_i =< k_1 : b_1, k_2 : b_2, ..., k_n : b_n >$. The problem entails that we identify frequent sequence patterns of attributes or subtrees of the transactions. This breaks down to finding itemset inclusion and structural inclusion [8].

Definition 1 (itemset inclusion). Given $T_i = (V_i, E_i, \lambda_i)$ where λ is a function that maps an itemset to each node. $T_1 = (V_1, E_1, \lambda_1)$ is included in $T_2 = (V_2, E_2, \lambda_2)$ denoted by $T_1 \sqsubset_I T_2$ if $V_1 = V_2$ and $E_1 = E_2$ and $\forall v \in V_1, \lambda_1(v) \subseteq \lambda_2(v)$.

Definition 2 (structural and itemset inclusion). Given $T_i = (V_i, E_i, \lambda_i)$ where λ is a function that maps an itemset to each node. $T_1 = (V_1, E_1, \lambda_1)$ is included in $T_2 = (V_2, E_2, \lambda_2)$ denoted by $T_1 \sqsubseteq T_2$ if T_1 is an isomorphic subtree of T_2 such that $\exists$ a mapping σ such that $T_1 \neq T_2$ and $\forall(u, v) \in E_1, \exists(\sigma(u), \sigma(v)) \in E$ and $\forall x \in V_1.\lambda_1(x) \subseteq \lambda_2(x)$.

3. SOLUTION

To solve the problem stated above, we have developed the FSP-Scop algorithm. This algorithm takes as parameters a Dataset and the mininmum support count. Due to the unique and complex nature of the problem, the algorithm generally solves it by combining two traditional data mining techniques. These are frequent itemset mining and sequence pattern mining. The FSP-Scop algorithm works in 3 main phases. We will give a brief overview of the phases and describe them into more details in the next subsection.

In the first phase, we do a frequent itemset mining to reduce the set of attributes or items of each node to only the frequent items or frequent attributes in the node. For each node, all the items in the node are listed out and their support counts are obtained. If the support count for a particular item is below the minimum support count value, the item is discarded from the particular node. To do this, we used the fp-growth algorithm which is one of the fastest frequent itemset mining algorithm.

The second phase consists of redcuing the tree to a sequence. To do this, we used the pre-order-string encoding as defined in the preliminaries section. Each node or record in in the dataset is mapped to their unique record ids. After this we then apply the pre-order encoding system to get the sequence. More details we will be given in the subsequent section.

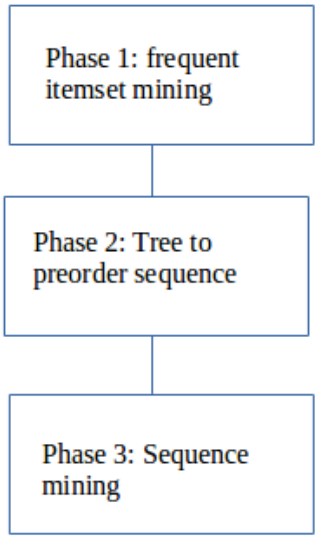

Figure 2: Phases of the FSP-Scop algorithm

Finally, a sequence mining algorithm is used to mine the frequent sequence patterns in the reduced tree. We mine for concise closed subsets in order to be able to generate induced subtrees. We use the Gap-bide sequence mining algorithm to achieve this. After obtaining the frequent pattern sequences, The nodes are then mapped back to their original values to show the results more clearly. The FSP-Scop algorithm we have developed mines for closed sequence patterns. A closed sequnce pattern is a sequence pattern in which no superset of that particular sequence has the same support count. A formal definition is given in the subsection explaining phase 3 of the algorithm [6].

3.1 Phase 1 Frequent Itemset mining

In the first phase of the algorithm we mine for frequent itemsets in order to find frequent items for the entire data as stated above. We then reduce each record to its most frequent itemset. Nodes or records for that matter without a frequent itemset are dicarded altogether. The fp-growth algorithm was crucial for this. The dataset and the minimum support sount are the inputs needed for this phase.

The data is first scanned to get a list of all the individual items. We then obtain the support count obtain the support count for each item. After this, a second pass is performed through the data. In this pass every record is read and infrequent items are removed from the record. The items in the record are sorted in ascending order of their frequencies. This is necessary for building the fp-tree [4]. An example illustrating this is process is shown in Figure 3.

To build the fp-tree, we start from the root node which is

a d f		d a
a c d e		d c a e
b d		d b
b c d		d b c
b c		b c
a b d		d b a
b d e		d b e
b c e g		b c e
c d f		d c
a b d		d b a

d	8
b	7
c	5
a	4
e	3
f	2
g	1

Figure 3: Deleting infrequent items and sorting items in record. original dataset on the left. Filtered and sorted items on the right. min_sup=3

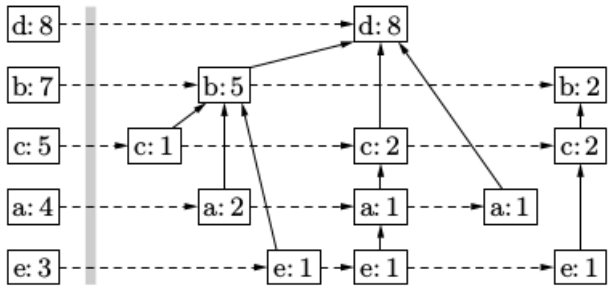

Figure 4: An implementation of the fp tree.

null in value. Each node of the fp-tree is made up of the name of the item and the count of the item. For each record in the data, the first item is added to the tree so that it becomes the child of the root node. The next item in the record becomes the child of its predecessor until the last item in the record is added. The same process is repeated for the next record. However, in the next record, if the first item already exists in the tree, the count of the item is increased without starting a new branch from the node. This is done for the second item also. If the second item is also the same, the count is increased. This continues until the next item is not a child of its predecessor in that particular branch, in which case a new branch is started from that point. A header table is kept to store the total count of a particular item in the tree. Figure 4 is an example of an implementation of the fp-tree using the data in Figure 3. the boxes on the left represent the header table[4].

After the fp-tree has been built, mining the frequent itemset becomes very easy and can be done without reading the original data which is one of the benefits of using the fp-growth approach for this phase. To mine for frequent itemsets, the divide and conquer approach is used through a bottom-up approach using the fp-tree as described in [4]. The pseudocode for this process is outlined in algorithm 1 and algorithm 2.

Input: Dataset D, minimum support min_sup
Output: List of frequent itemsets
Function `FP_growth`(D, min_sup)**:**
  Get list of all records from D
  foreach $record \in D$ **do**
    foreach $item \in record$ **do**
      if $item\_support\_count < min\_sup$ **then**
        discard $item$ from $record$
      end
    end
    sort $items$ in $record$ in ascending order of item support count
  end
  $S \longleftarrow setofrecords$
  Call $construct\_tree(setofnodesS, min\_sup)$
  foreach $transaction \in S$ **do**
    scan tree to get frequent itemsets
  end
End Function

Algorithm 1: Algorithm for FP-growth

Input: Set of nodes or records in D S
Output: FP tree
Function `construct_tree`(S)**:**
  foreach $record \in S$ **do**
    foreach $item \in record$ **do**
      if *item does not exit in current tree branch* **then**
        create a path $null \rightarrow i_{i_1} \rightarrow i_{i_2} ... \rightarrow i_{in}$
      else
        update existing dode
      end
    end
  end
  update *count* for each item in tree Store total count for each item in header table
End Function

Algorithm 2: Algorithm for Constructing FP tree

3.2 Phase 2 Reducing Tree to Sequence

In this phase, we use the pre-order-string tree reduction technique used in [9] . Before reducing the tree to a string sequence, the representation of the tree has to be modified to make the process easier. In the original problem, the data is a set of transactions where each transaction is a set of records containing attributes. each record can be uniquely represented with an integer id. We thus, map and represent each node with its id as shown in Figure 5 and Fiqure 6. We then use the pre-order traversal method to encode the tree. We start from the root node and move to its leftmost child, this process is continued until a leafnode is reached. When a leaf-node is reached, we backtrack to its parent node. We use a special character known as the end flag to indicate a backtrack. As noted in the introduction, we use '$' to represent a backtrack in this algorithm. With this process we obtain a string encoding of the algorithm. An example of this encoding is shown in Figure 1. With this format, we can use a pattern sequence algorithm to mine for frequent substructures or subtrees in the data. The process is outlined in Algorithm 3 and Algorithm 4.

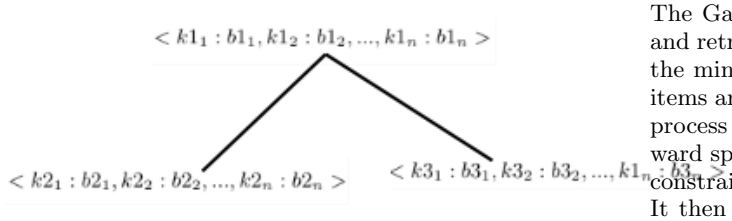

Figure 5: tree with record nodes

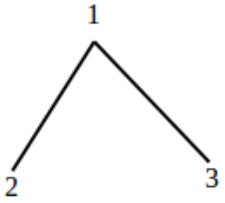

Figure 6: reduced tree in figure 6

Input: Set of trees D
Output: Sequence of items
Function `to_sequence(`D`)`**:**
  foreach $tree \in S$ **do**
    $curr\_node \longleftarrow root$
    Call $traverse()$
  end
  $S^* \longleftarrow S^* \cup S$ **return** S^*;
End Function

Algorithm 3: Convert tree to item sequence

Function `traverse()`**:**
  $S \longleftarrow null$
  $S \longleftarrow S + leftmostnode$
  $curr\_node \longleftarrow leftmostnode$ **if** $node == leaf$ **then**
    $S \longleftarrow S \cup \$$
    backtrack to parent node
    if *has right node* **then**
      $S \longleftarrow S \cup rightnode$
      $curr\_node \longleftarrow rightnode$
      Call $traverse()$
    end
  end
  return S
End Function

Algorithm 4: Algorithm for tree traversal

3.3 Phase 3 Sequence Pattern Mining

In this phase, the frequent sequences are mined to obtain the frequent substructures. To implement this phase, we used the Gap_BIDE algorithm. A sequence $S_a =< a_1, a_2, ...a_n >$ is cotained in $S_b =< b_1, b_2, ...b_n >$ if $n \leq m$ and $\exists 1 \leq i_1 < i_2 < ... < i_n \leq m$ such that $a_1 = b_{i_1}, a_2 = b_{i_2}, ..., a_n = b_{i_n}$ [6] similar to Definition 2. The Gap_BIDE algorithm is very suitable for closed gap constrained sequential patterns. Given a pattern P', P' is said to be closed gap constrained if $\nexists$ contiguous superpattern $P''ofP'$, where $seq\_sup^{SDB}(P') = seq\_sup^{SDB}(P'')$ [6] similar to the closed sequence pattern previously explained. SDB refers to sequential database.

The Gap_BIDE algorithm starts by scanning the database and retrieves a set of all the frequent items of length 1 using the minimum support count as the threshold value. These items are used as the prefixes for the sequence mining. This process is outlined in Algorithm 5. It then scans the backward spaces of the prefix and uses a method called the gap-constrained Back-scan pruning to prune the search space [6]. It then scans the forward spaces to ensure that the prefix pattern is closed. A closed pattern is that in which there exists no proper superpattern[8]. The prefix is returned as a gap-constrained sequential pattern. It scans the forward space of every instance of the prefix pattern and gets a set of all locally frequent items. These items are used to extend the prefix. It iterates this process to get new gap-constrained sequential patterns for the new prefix [6]. This process is also outlined in Algorithm 6. The algorithm returns a set of all the frequent gap-constrained closed sequential patterns together with their support counts.

3.4 Finding Frequent patterns

Once the above phases have been completed, the task is to output the frequent patterns of attributes. Firstly, note that the sequences are represented using the node ids. We therefore need to convert the sequence back to their respective itemsets. The ids are used to map each node back to the actual representation. With this done, we can easily read the frequent patterns of attributes together with their support counts

Input: Sequence data S
Output: set of closed gap-constrained sequential patterns
Function `Gap_Bide(`$S, min\_sup, MandN$`)`**:**
  $Ls \longleftarrow allfrequentsequencesoflength1inS$
  foreach $item \in Ls$ **do**
    call Pattern_growth($item$)
  end
  return
End Function

Algorithm 5: Algorithm for Gap_BIDE

Input: item
Output: set of closed gap-constrained sequential patterns with the input as prefix
Function `Pattern_growth(`$Prefix$`)`**:**
  Scan backward spaces of $Prefix$ Prune search space
  Scan forward spaces of $prefix$ **if** $prefixclosed$ **then**
  end
  output $prefix$
  Scan for all appearances of $prefix$
  $X \longleftarrow setoflocallyfrequentitems$
  foreach $item \in X$ **do**
    $prefix' = prefix + item$
    call Pattern_growth($prefix'$)
  end
  return
End Function

Algorithm 6: Pattern growth algorithm

4. EXPERIMENTS

To be able to validate and evaluate the algorithm, we had to carry out an implementation on a realistic dataset to show how well the algorithm works and how it performs in comparison to other algorithms. In this section, we will first

```
<tweet id="27688655912975649">
    <categories category= university>
        <topic>Hoping to study at UC Berkeley</topic>
        <content>I love Berkeley but every time </content>
    </categories>
    <entities entity=The University of California, Berkeley
        <timestamp>2012-12-7-04:11</timestamp>
    </entities>
</tweet>
```

Figure 7: XML format of dataset

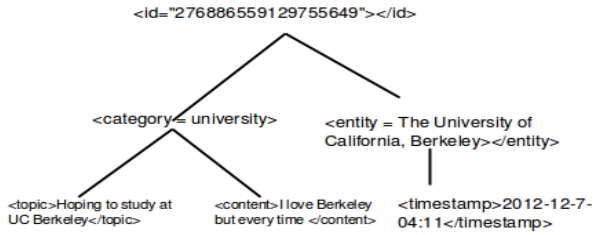

Figure 8: tree representation of Figure 7

briefly describe the implementation process and the tools used, we will look at the dataset used for the evaluation and finally, we will look at the various experiments conducted.

4.1 Environment and implementation

To implement the algorithm, we used python 3. We used python libraries like the ete3 library to help visualise the tree and to process the tree. We primarily developed the algorithm and carried out the experiments on an Intel(R) Core(TM) i3-6006U Processor @ 1.8GHz with 4GByte main memory running a Linux Ubuntu 16.04 operating system. The experiments were carried out mainly using Jupyter notebook together with the Ubuntu terminal.

4.2 Dataset

To properly evaluate the algorithm, we needed a dataset that exactly fits the problem being solved. We used the 2014 Topics in xml format (INEXTC2014_topics) which is publicly available on the INEX website. This is an XML file containing tweets collected by INEX. However, we needed to preprocess this data to convert it to a suitable format for the algorithm to process. We used the newick format as the input to the algorithm. We used the newick format because the ete3 library on python supports newick format for visualisation and processing. We again had to tweak the data to give it a more nested format to properly test the depth of the algorithm. The data consists of tweets from various sources and covering diverse topics. Each tweet is made up of a tweet id tag, a category tag, an entity tag, a topic tag and a timestamp tag. The XML file contains a total of 216 tweet. We treat each tweet in the XML as a unique transaction and for that matter a tree in this context. The nodes of the tree are the tags of the XML file for each tweet as listed above together with the value they contain. An example is shown in Figure 7. The tree structure is illustrated in Figure 8.

In each node, we took the name of the tag and the content of the tag to be the attributes of the tree so that each node is a tuple $(tag\_name, tag\_content)$. Due to the constraint in the use of symbols of the newick format, we could not use the comma as a separator in the tuple so the character '|' is used as a separator in place of the comma. To represent the data in newick format, nodes are separated by a comma and a child of a node is put in a bracket right next to the parent node. The newick format for Figure 7 is shown in Figure 9. This is the format with which the algorithm works. The dataset we used is thus a file containing 216 newick

```
(@id|276886559129755649,
    (category|university,
        (topic|Hoping to study at UC Berkeley,
        content|I love Berkeley but every time),
    entity|The University of CaliforniaBerkeley,
        (timestamp|201212704)));
```

Figure 9: Figure 8 in newick format

trees. Each row in the dataset represents a tree which is a tweet together with all its details. The newick tree can be visualised using the ete3 python library. An image of the visualisation is shown in Figure 10.

The output of the algorithm is a tuple containing the frequent pattern in the first index, the second index contains the support count. An example of this is also shown in Figure 11.

4.3 Experimental results

We performed the experiment using different sizes of data in order to see how well the algorithm scales. In the first instance we used 54 tree from the dataset, the second dataset was contained 108 trees and the last dataset was the full data of 216 trees. To evaluate the algorithm, we tested for how the algorithm behaved against metrices like execution time and number of patterns retrieved.

In the first experiment we run all the three algorithms with different support counts and measured the execution time. For the smallest data of 54 trees, we started with a minimum support count of 2. This recorded an execution time of 0.27 seconds. We then increased the minimum support count to 10 which run in 0.20 seconds, We doubled the minimum

```
   /-@id|276886559129755649
  |
  |   /-category|university
--|  |
  |  |   /-topic|Hoping to study at UC Berkeley
  |  |--|
   \-|   \-content|I love Berkeley but every time Im h
b
     |
     |--entity|The University of CaliforniaBerkeley
     |
      \- /-timestamp|201212704
```

Figure 10: visualisation of newick tree

```
([['timestamp']], 183),
([['entity']], 149),
([['topic', 'sales']], 118),
([['category', 'automotive'], ['topic', 'sales']], 107),
([['@id']], 107),
(['$', ['timestamp']], 106),
(['$', ['entity']], 85),
([['entity', 'Audi']], 82),
([['timestamp'], ['entity', 'Audi']], 81),
([['@id']], 80),
(['$', '$', ['timestamp']], 75),
(['$', ['@id']], 74),
([['entity']], 68),
([['topic']], 52),
([['@id']], 52),
([['entity'], ['topic']], 50)]
```

Figure 11: output format

support count to 20 which took 0.211 seconds, minimum support count of 50 took 0.244 seconds and minimum support count of 100 ran in 0.215 seconds. From this analysis we realised that the execution time kept reducing almost exponentially until it hit an asymptotic value of about 2.3 in which case no frequent sequence pattern existed. The other datasets we used followed a similar pattern. For instance the dataset with 108 trees started at 0.79 seconds and reduced to an asymptotic execution time value of about 0.57 seconds at minimum support count of 50. Same was observed for dataset size 0f 216 trees which started at 2.26 seconds and hit an asymptotic value of about 1.80 seconds at minimum support count of 100. A diagram showing a visualisation of the result of this experiment is in Figure 12.

The second experiment was conducted to measure the variation in the number of patterns for the three sets of data as the minimum support count varried. Again, for the smallest set of 54 trees, we started from a minimum support count of 2. This generated 18 frequent sequence patterns. A minimum support count count of 10 generated only 2 frequent sequence patterns for the same dataset.At a minimum support count of 20, 2 frequent sequence patterns were generated. Same number was generates for min_sup count of 50 and none for minimum support count of 100 since the total number of trees in the dataset was less than the minimum support. For the second dataset of trees of 108 trees, the number of frequent sequence patterns for the same minimum support counts were 35, 5, 4, 3 and 2 respectively. It had no frequent sequence pattern for minimum support count of 150. For the full dataset, the frequent sequence patterns for the same minimum support counts plus an additional test for a minimum support count of 150 resulted in 58, 10, 7, 5, 2, 2 respectively. We realised that the frequent sequence patterns also decayed exponentially with increasing minimum support count. A visualisation of this is in Figure 13.

Table 1 contains the most frequent sequence patterns our algorithm mined from the whole dataset.

4.4 Baseline Comparison

In order to give a proper assessment of the function of the algorithm, it was necessary to carry out an evaluation of the algorithm. However since the nature of the problem was such that it did not involve training a model where we could do an evaluation with a test set after training with a training data, we came up with a series of tests that enabled us to do

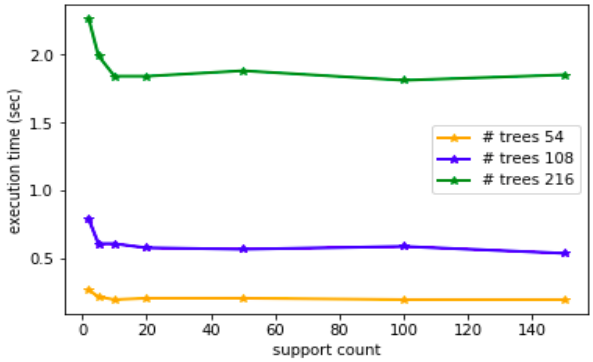

Figure 12: time per min_support

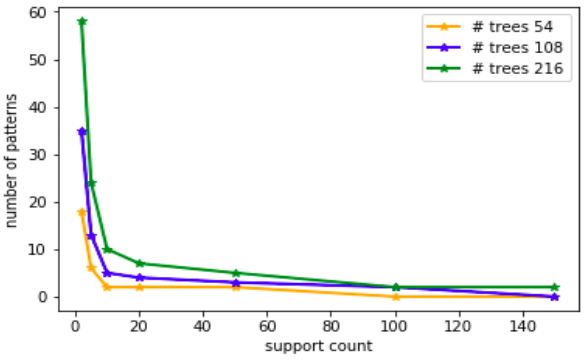

Figure 13: number of patterns per min_support

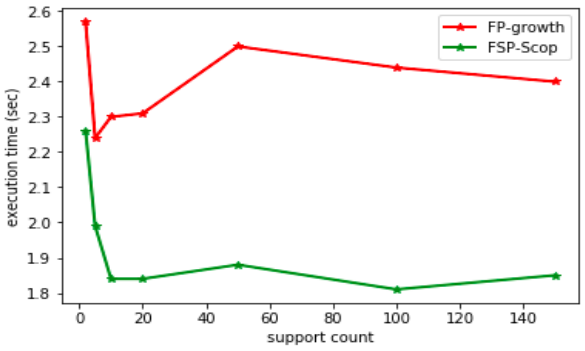

Figure 14: time per min_support in baseline test

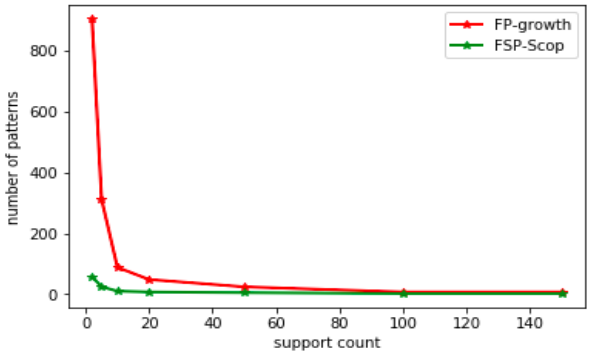

Figure 15: number of patterns per min_support in baseline test

a fair assessment. One of the key steps we took was to compare our algorithm with a traditional data mining algorithm. For this experiment, we used the FP-growth algorithm. Notice that the FP-growth algorithm was used as part of our implementation to get frequent itemsets; however, in this case we used the FP-growth as a Frequent itemset tree mining technique for complex objects. It is worth noting that FP-growth performs poorly as a sequence pattern mining algorithm for frequent itemsets but for the purpose of getting a baseline to show the performance of FSP-Scope, we saw it as a useful yardstick.

Firstly, we did an execution time comparison on both algorithms on different support counts: 2, 5, 10, 20, 50, 100, 150. For which our algorithm substantially outperformed the FP-growth. However, we bear in mind that the FP-growth algorithm is not very impressive in itself for tree-like sequence mining of frequent itemsets. The FP-growth algorithm recorded execution times of 2.57, 2.24, 2.30, 2.31, 2.5, 2.44 and 2.40 seconds for the respective minimum support counts. This is captured in the graph in Figure 14.

Next, we did a comparison to show the number of patterns generated for various minimum support counts. Here, The FP-growth algorithm seems to generate much more sequences than the FSP-Scop, however a proper look at the result it generates shows these patters are not in sequential order. With this, we are able to claim that the FSP-Scop algorithm performs better in terms of generating patterns in sequence. Again, because only closed itemset sequence patterns are mined by FSP-Scop, it reduces the amout of memory needed to generate the itemset sequences. Figure 15 captures the number of patterns generated by both algorithms.

To further test our evaluation, we created a toy dataset by duplicating just 4 trees in the original dataset. The trees were repeated 10 times each with the exception of one tree which was duplicated five times hence, totaling 35 trees. Our algorithm worked satisfyingly in this test. The result of this test is shown if Table 2. The FP-growth algorithm on the other hand generated a lot of false negative results. This again explains why the FP-growth algorithm generates many more itemset sequences. When our algorithm was however tested on datasets containing numbers, the performance of the algorithm reduced significantly. This was

Table 1: some frequent sequence patterns mined from the dataset with FSP-Scop

Frequent Patterns	Support count
'timestamp'	216
'id'	215
'id'⟶ 'category': 'university'	88
'category': 'automotive"	74
'id'⟶ 'category': 'automotive'	73
'id'⟶ 'category': 'banking'	44
'entity': 'YaleUniversity'⟶ 'timestamp'	25
'entity': 'HarvardUniversity'→ 'timestamp'	11
'id'⟶ 'category': 'music'	10
'entity'⟶ 'timestamp'	10

however as a result of a problem with the implementation rather than a flaw in our algorithm itself.

Table 2: some frequent sequence patterns mined from toy experimental dataset with FSP-Scop

Frequent Patterns	Suppt cnt
'timestamp'	35
['id'] ['category', 'automotive']	35
['id']['category': 'automotive']['content', 'Badnews']	20
['entity', 'ChryslerGroupLLC'], ['timestamp']	15
['id'], ['category', 'automotive'] ['content', 'Badnews'] ['content', 'Badnews']['entity', 'ABVolvo'], ['timestamp'],]	10
['id'],['category', 'automotive'], ['content', 'Badnews'] , ['content', 'AudiIndiasoldcars'] ,['entity', 'AUDIAktiengesellschaft'], ['timestamp'],]	10
['id'], ['category', 'automotive'], ['topic', 'Complaints'], ['content', 'Americaifyouwant'], ['entity', 'ChryslerGroupLLC'], ['timestamp']	10

5. CONCLUSION

In this work, we discussed the problem of Frequent itemset mining in tree-like sequences of complex objects. We looked at the relevance of finding a practical solution to this problem. We then reviewed some existing works and algorithms related to this topic. We introduce our solution to solve this problem which is the FSP-Scop algorithm. In the final section of this work, we carried out various tests to evaluate the algorithm. This algorithm has a few limitations which can be addressed to improve its performance. Firstly, the algorithm can be optimized to improve its execution time.

Again, this algorithm can be extended to mine for frequent itemset sequences that are not closed depending on the application scenario.

6. REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Acm sigmod record*, volume 22, pages 207–216. ACM, 1993.

[2] C. Borgelt. An implementation of the fp-growth algorithm. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, pages 1–5. ACM, 2005.

[3] C. Garboni, F. Masseglia, and B. Trousse. Sequential pattern mining for structure-based xml document classification. In *International Workshop of the Initiative for the Evaluation of XML Retrieval*, pages 458–468. Springer, 2005.

[4] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM sigmod record*, volume 29, pages 1–12. ACM, 2000.

[5] S. Hido and H. Kawano. Amiot: induced ordered tree mining in tree-structured databases. In *null*, pages 170–177. IEEE, 2005.

[6] C. Li and J. Wang. Efficiently mining closed subsequences with gap constraints. In *proceedings of the 2008 SIAM International Conference on Data Mining*, pages 313–322. SIAM, 2008.

[7] Y. Miyoshi, T. Ozaki, and T. Ohkawa. Frequent pattern discovery from a single graph with quantitative itemsets. In *Data Mining Workshops, 2009. ICDMW'09. IEEE International Conference on*, pages 527–532. IEEE, 2009.

[8] C. Pasquier, J. Sanhes, F. Flouvat, and N. Selmaoui-Folcher. Frequent pattern mining in attributed trees: algorithms and applications. *Knowledge and Information Systems*, 46(3):491–514, 2016.

[9] L. Zou, Y. Lu, H. Zhang, R. Hu, and C. Zhou. Mining frequent induced subtrees by prefix-tree-projected pattern growth. In *Web-Age Information Management Workshops, 2006. WAIM'06. Seventh International Conference on*, pages 18–18. IEEE, 2006.