Generalized Sequential Patterns

Tomasz Gebarowski and Adam Szczepankiewicz

***Abstract*—The aim of this paper is to sum up the project related to implementing and investigating the behavior of Generalized Sequential Pattern algorithm. Apart from technical details related to GSP algorithm itself and its implementation, the paper explores its operation and returned results when modifying algorithm configuration parameters such as min/max gap and window size.**

***Keywords*—Data Mining, GSP, Generalized Sequential Pattern, Sequential Patterns**

I. Introduction

GSP[1] Algorithm (Generalized Sequential Pattern algorithm) is an algorithm used for sequence mining.

The algorithm itself makes multiple database passes. In the first pass, all single items are counted. From frequent single items, a set of candidate 2-sequences is formed, resulting in another pass required to identify their frequency. The frequent 2-sequences are used to generate the candidate 3-sequences, repeating the process until no more frequent sequences are found.

There are two main steps in the algorithm:

- Candidate Generation. Given the set of frequent (k-1)-frequent sequences F(k-1), the candidates for the next pass are generated by joining F(k-1) with itself. A pruning phase eliminates any sequence in which at least one of subsequences is not frequent.
- Support Counting based on hash tree search to speed up process of determining support.

As opposed to other sequence mining algorithms, the GSP algorithm discovers frequent sequences taking into account minimum and maximum gap among the sequence elements. Apart from that due to sliding window parameter it is possible to define a time interval within which items are observed as belonging to the same event.

II. Assumptions and Design

The GSP algorithm itself has been implemented using C++ and object oriented design patterns. Figure 1 depicts a class diagram reflecting interconnections of classes and structure of the project. It is worth noticing that for the convenience sake the concepts of Itemset and Sequence are modeled as individual classes. The core part of the project is the GspAlgorithm class which stores references to two GspSequencePool objects. The first one comprises the frequent k-sequences found in the previous pass, while the other one contains the (k+1)-candidates to be used in the next pass of the algorithm. The GspSequenceReader class defines a common interface for input data reading functionality. Depending on the needs one may provide data from file or directly from array (testing purposes) using specific interface implementations. In the final version of the program the GspFileReader class is used to extract sequences from a simple data file.

The algorithm starts by making a first pass over an input file to generate 1-frequent sequences, which are stored in the GspSequencePool object. After that, the algorithm iterates, making passes over the data using the following procedure:

- The frequent sequences from the previous pass are used to generate a new GspSequencePool object with the candidates of (k+1)-length. For this purpose the frequent set is represented as a suffix tree in a GspJoinTree object.
- The candidate set is pruned to eliminate the sequences containing non-frequent contiguous subsequences [1].
- The remaining candidate sequences are stored in a hash tree (GspHashTree).
- The input file is passed and every data sequence is matched with its possible subsequences by traversing the previously created tree. Finally, it is checked which of those candidates are supported by a real data sequence. For this purpose each sequence is represented as a table of linked list timestamps marking the occurrences of each item using the GspItemsetIterator and GspItemSetIterators objects.
- Candidates with the support value below the threshold value are dropped.
- Remaining sequences are considered a new frequent set for the next pass.

It has to be noted that the algorithm can stop at any point of the procedure. It happens if the candidate set becomes empty after an operation. Then the current frequent set is considered a solution.

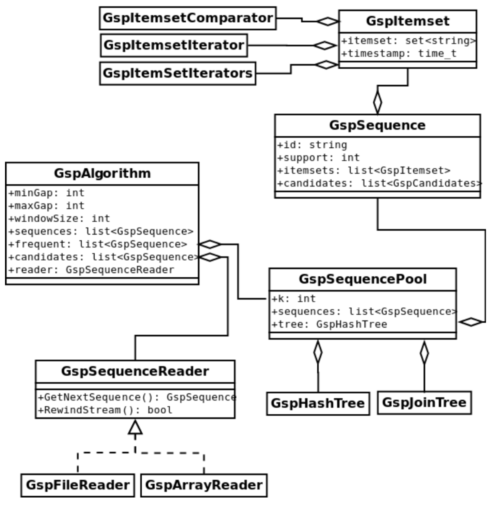
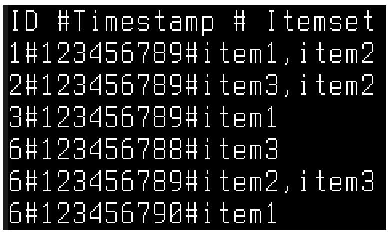


Fig. 1. Project class diagram

III. Input and output data format

Figure 2 depicts an overview of data format used by GSP algorithm implemented for the purpose of the project.



```
ID #Timestamp # Itemset
1#123456789#item1,item2
2#123456789#item3,item2
3#123456789#item1
6#123456788#item3
6#123456789#item2,item3
6#123456790#item1
```

Fig. 2. Input data format

Data series are in tabular form, where individual columns are separated by a hash symbol. First column of the row corresponds to a unique identifier of the user, second to the timestamp of transaction and the last to a list of comma separated items present in the transaction.

The output has a form of a simple text file. Each line corresponds to a frequent sequence. Each transaction is printed inside the brackets with the items separated by a space character:

```
{ 100 301 }{ 100 }{ 101 317 }{ 307 }
{ 100 307 }{ 100 }{ 101 317 }{ 307 }
```

IV. User guide

The resulting program is called *gsp* and is a command-line tool. Some arguments have to be specified for its successful execution:

```
./gsp input_file output_file min_sup
        window_size min_gap max_gap
```

The arguments have to be specified in given order and their meaning is as following:

- *input_file* – the path to the input file.
- *output_file* – the path to the output file for the result sequences. It has to be a writable location. If existing before the operation, the file will be overwritten.
- *min_sup* – the minimum support. A sequence will be considered frequent if its support is larger than the specified value.
- *window_size* – the GSP algorithm sliding windows size.
- *min_gap* – the GSP algorithm minimum gap size.
- *max_gap* – the GSP algorithm maximum gap size.

V. Sample data set

The program was run for a sample data set containing two sequences:

```
1#2#1,2,
1#6#1
1#9#1,2
2#2#2
2#9#1,2
2#11#1
```

The values for the parameters minimal support,window size, minimum and maximum gap were 1, 0, 0, 0 respectively. The resulting frequent sequences were:

```
{ 2 }{ 1 }{ 1 }
{ 2 }{ 1 2 }
{ 1 2 }{ 1 }
```

VI. Results

The algorithm was run for 3 data sets containing 37, 364 and 3651 data sequences. In each case the the minimum support was set to a value corresponding to 95% of the overall sequences, the minimum gap was 12 and the maximum gap was 15. Two tests were performed whit different values of the sliding window parameter. In the first case it was set to 0 and in the second it was set to 1.

Support	Window	Time elapsed	Sequences	Length
35/37	0	0.22 s	1	9
344/364	0	14.25 s	2	9
3450/3651	0	164.55	3	9
35/37	1	1.32 s	3	12
344/364	1	76.64 s	3	11
3450/3651	1	828.22	3	12

TABLE I
Results of tests.

As expected, the execution time increases for larger data sets. Also adding the sliding window increases the time needed to perform the operation due to larger number of candidates that meet the minimal support.

VII. Conclusions

The implemented GSP algorithm proved its correctness by analyzing various small input datasets and comparing the results with straightforwad evaluation. Primarily, the tests with larger input data failed, due to lack of implementing candidates pruning and rather naive method of joining sequences. After implementing candidate pruning as defined in [1] and using suffix tree for joining sequences the performance increased significantly. Anyhow, even after making these enhancements, the tests involving very large datasets were time consuming.

Finally, in the analyzed datasets most of the sequences were frequent, it resulted in high memory consumption when processing the data sets. For some program settings the operation could not be finished due to lack of memory. It is caused by huge amounts of candidate sequences generated even for very high minimal support values.

All in all, both the implementation of GSP algorithm and performed tests proved its usefullness and the algorithm procedures defined in [1] are correct.

References

[1] R. Srikant and R. Agrawal, “Mining sequential patterns: Generalizations and performance improvements,” IBM Almaden Research Center, Paper, 1995.