University of Dhaka

Department of Computer Science and Engineering

CSE-4255 : Introduction to Data Mining and Warehousing Lab

Lab Report: Comparative Analysis of GSP and PrefixSpan Algorithm

Submitted By:

Name : Zisan Mahmud

Roll No : 23

Submitted On:

MAY 29, 2025

Submitted To:

Dr. Chowdhury Farhan Ahmed

Md. Mahmudur Rahman

Contents

1 ABSTRACT

This report presents a comparative analysis of GSP and PrefixSpan algorithm. Total four datasets (BMSWebView1, BIKE, SIGN, E-Shop) are used and both algorithms are implemented on the datasets at various support levels to determine the most efficient method. The time and memory usage complexity of the two algorithms is also analyzed.
keywords: GSP(Generalized Sequential Pattern), PrefixSpan.

2 INTRODUCTION

Sequential pattern mining is an important task in data mining since it entails uncovering statistically relevant patterns where values or events are given in a specific order. This has numerous uses, including market basket analysis, web clickstream analysis, DNA sequencing, and more. Two of the most well-known algorithms for this purpose are GSP (Generalized Sequential Pattern) and PrefixSpan (Prefix-projected Sequential Pattern Mining).

2.1 GSP

The GSP algorithm is a traditional sequential pattern mining technique that uses a candidate generation-test strategy. It is a generalization of the Apriori concept where longer sequences are progressively generated from frequent subsequences obtained in the earlier iterations. At every iteration, GSP scans the entire database to measure the support of the candidate sequences. It only stores sequences fulfilling the minimal support for further extension. It is possible to find all frequent sequential patterns using this method, but the enormous set of candidates, as well as the number of database scans, can make the process computationally costly.

2.2 PrefixSpan

The PrefixSpan algorithm is a pattern-growth approach to mine sequential pattern. PrefixSpan projects the database based on frequent prefixes and recursively grows the patterns by exploring the projected sub-databases. Each projection reduces the search space, and patterns are extended only when they meet the minimum support criterion. This mechanism eliminates the need for exhaustive candidate generation and multiple full database scans, leading to significant performance gains, especially for large or dense datasets.

3 IMPLEMENTATION DETAIL

This section outlines the implementation specifics for both the GSP and PrefixSpan algorithms, including data handling, core processes, and performance measurements.

3.1 GSP

Data Handling and Preprocessing: The input data consists of sequences of itemsets, where each itemset represents a collection of items occurring simultaneously. The `load_data` function is responsible for parsing the input file. The file format assumes space-separated integer tokens. A token of `-1` signifies the end of an itemset within a sequence, and `-2` marks the end of a complete sequence. The function reads the file line by line, accumulating items into an `current_event` (an itemset, represented as a Python set to handle unordered items within an event and automatically manage duplicates). Upon encountering a `-1`, the sorted list of items in `current_event` is appended to the `current_sequence`. When a `-2` is encountered, the `current_sequence` (if non-empty) is added to the main list of `sequences`. This process ensures that each sequence is a list of sorted itemsets.

Core GSP Algorithm: The GSP algorithm is implemented through several key functions:

- **Candidate Generation:** Candidate generation is a two-step process: joining and pruning.
  - **Initial Candidates (1-sequences):** The process begins by identifying all unique items across all sequences. Each unique item forms an initial 1-sequence (e.g., `[[item]]`). Their support is counted, and those meeting the minimum support threshold (`min_sup`) form the frequent 1-sequences (L_1).
  - **Generating k-sequences from (k-1)-sequences ($k > 1$):** The `gen_cands` function generates candidate sequences of length k from frequent sequences of length $k-1$ (L_{k-1}). For $k = 2$, it generates candidates by either combining two frequent 1-items into a single 2-itemset (e.g., `[[a, b]]`) or as two separate 1-itemsets (e.g., `[[a], [b]]`). For $k > 2$, the `gen_cands_for_pair` function is used. It attempts to join two frequent $(k-1)$-sequences, s_1 and s_2. A join is possible if the sequence obtained by removing the first item of s_1 is identical to the sequence obtained by removing the last item of s_2. If they can be joined, the new candidate c is formed by extending s_1 with the last item of s_2. If the last itemset of s_2 contains multiple items, only its last item is appended to the last itemset of s_1. If the last itemset of s_2 is a single item, it is appended as a new itemset to s_1.

- **Support Counting:** The `count_support` function determines the frequency of a candidate sequence (`cand_seq`) within the dataset (`sequences`). It iterates through each sequence in the dataset and utilizes the `is_subsequence` helper function. The `is_subsequence` function checks if a `subsequence` is present in a `main_sequence`. It employs a recursive approach: for each element (itemset) in the `subsequence`, it searches for a matching superset element in the `main_sequence` from the position following the previous match. An itemset I_s from the subsequence matches an itemset I_m from the main sequence if I_m is a superset of I_s.

- **Pruning:** After generating candidates, a pruning step is applied using the `prune_cands` function. This function leverages the Apriori principle: if a sequence is frequent, all of its subsequences must also be frequent. For each candidate sequence c generated, the `gen_direct_subsequences` function is called to create all its immediate $(k-1)$-subsequences. A $(k-1)$-subsequence is formed by removing a single item from one of the itemsets of c, or by removing an entire itemset if it contains only one item. The candidate c is kept only if all its generated $(k-1)$-subsequences are present in the set of frequent $(k-1)$-sequences (L_{k-1}).

Iterative Process: The algorithm iteratively generates frequent sequences:

1. L_1 = frequent 1-sequences.
2. For $k = 2, 3, \ldots$ until no more frequent sequences are found:
   (a) Generate C_k (candidate k-sequences) from L_{k-1} using `gen_cands`.
   (b) Prune C_k using `prune_cands` to get C'_k.
   (c) Count support for each candidate in C'_k.
   (d) L_k = candidates in C'_k with support $\geq$ `min_sup`.

The `min_sup` can be provided as an absolute count or a fractional value, which is then converted to an absolute count based on the total number of sequences in the dataset. The final list of all frequent sequences across all levels is collected and sorted, primarily by support (descending) and secondarily by sequence length (descending, as a tie-breaker).

Performance Measurement: To evaluate the efficiency of the implementation, two key metrics are recorded:

- **Execution Time:** Measured using the `time` module. The start time is recorded before the GSP main loop begins, and the end time is recorded after all frequent sequences have been found. The difference provides the total execution time.

- **Memory Usage:** Measured using the `memory_profiler` library. The memory usage is sampled before the main GSP logic (`memory_usage()[0]`)

and again after its completion. The difference between these two values approximates the memory consumed by the algorithm.

Output and Reporting:

- `gsp_print` is used for verbose output, aiding debugging and analysis.
- It logs:
  - Candidate sets C_k (before and after pruning),
  - Frequent sets L_k for each level k,
  - Final summary of all frequent patterns.
- Output is sent to both the console and optionally a file.
- Sequences are formatted for readability (e.g., `<1 -> 2 3 -> 4>`).
- The main `gsp` function returns:
  - List of frequent sequences,
  - Memory usage,
  - Execution time.

3.2 PrefixSpan

Data Handling and Preprocessing:

- The input data for PrefixSpan consists of a collection of sequences, where each sequence is an ordered list of itemsets. Each itemset contains one or more items that occur together.
- The data is parsed such that each sequence is represented as a list of itemsets, and each itemset is a list or set of items. Special tokens (such as `-1` for end of itemset and `-2` for end of sequence) are used to delimit itemsets and sequences during parsing.
- The preprocessing step ensures that all sequences are consistently formatted and that empty itemsets or sequences are removed.

Core PrefixSpan Algorithm:

- **Recursive Pattern Growth:** PrefixSpan employs a recursive, pattern-growth approach. The algorithm starts with an empty prefix and recursively extends it by exploring all possible frequent items that can follow the current prefix in the projected database.

- **Frequent Item Discovery:** For a given prefix, the algorithm scans the projected database to find all items that can be appended to the prefix (either as a new item in the current itemset or as a new itemset) such that the resulting sequence remains frequent (i.e., its support meets or exceeds the minimum support threshold).
- **Database Projection:** For each frequent extension, the algorithm constructs a new projected database. This database contains the suffixes of sequences that match the current prefix, starting from the position after the prefix.
- **Support Counting:** The support of a sequence is determined by counting the number of sequences in the projected database that contain the sequence as a prefix.
- **Termination Condition:** The recursion terminates when no further frequent extensions can be found for the current prefix.
- **Result Collection:** All frequent sequential patterns discovered during the recursive exploration are collected and stored, typically along with their support counts.

Performance Measurement:

- **Execution Time:** The total execution time of the PrefixSpan algorithm is measured using system time functions. The start time is recorded before the mining process begins, and the end time is recorded after all frequent patterns have been discovered.
- **Memory Usage:** Memory consumption is monitored using profiling tools or libraries (such as `memory_profiler` in Python). Memory usage is sampled before and after the mining process to estimate the peak memory required.
- **Reporting:** Both execution time and memory usage are reported at the end of the algorithm, providing insights into the efficiency and scalability of the implementation.

Output and Reporting:

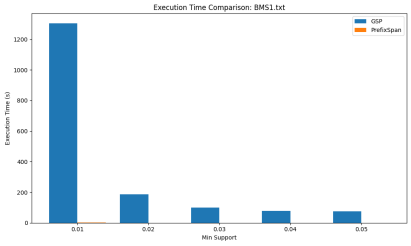
- The implementation outputs all discovered frequent sequential patterns, typically sorted by support or pattern length.
- For each pattern, the sequence and its support count are reported.
- Optionally, intermediate results and detailed logs can be written to a file or displayed on the console for analysis and debugging.

4 EXPERIMENTAL RESULT

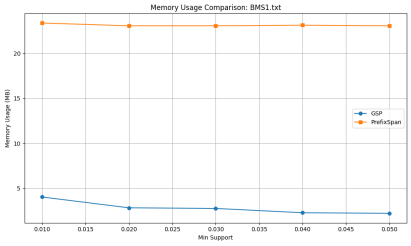
Here I present the experimental result of the implementation of GSP and PrefixSpan Algorithm.
Test were run on a Laptop with 2.30 GHz Intel(R) Core(TM) i7 - 11800H (11th Gen) processor and 8Gbyte Ram. The operating system was Windows 11.
The results are presented in the form of line graphs and bar graphs.

BMSWebView1

It has total 59,601 transactions, 497 distinct items and average sequence length is 2.42.



(a) Execution Time



(b) Memory Usage

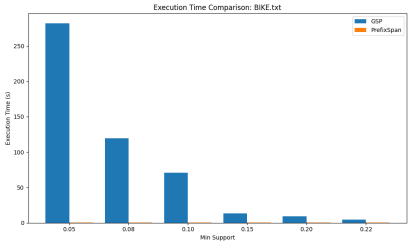
Min_sup	Metric	GSP	PrefixSpan
0.05	Runtime (s)	76.13	0.34
	Memory (MB)	2.24	23.05
	Frequent Patterns	4	4
0.04	Runtime (s)	77.35	0.35
	Memory (MB)	2.30	23.12
	Frequent Patterns	5	5
0.03	Runtime (s)	100.66	0.46
	Memory (MB)	2.78	23.05
	Frequent Patterns	11	11
0.02	Runtime (s)	185.79	0.59
	Memory (MB)	2.85	23.05
	Frequent Patterns	22	22
0.01	Runtime (s)	1303.56	1.10
	Memory (MB)	4.05	23.36
	Frequent Patterns	77	77

(c) Comparison Table

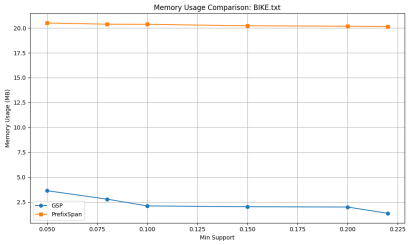
Figure 1: Performance Comparison of GSP vs PrefixSpan (BMSWebView1 Dataset)

BIKE

It has total 21,078 transactions, 67 distinct items and average sequence length is 7.27.



(a) Execution Time



(b) Memory Usage

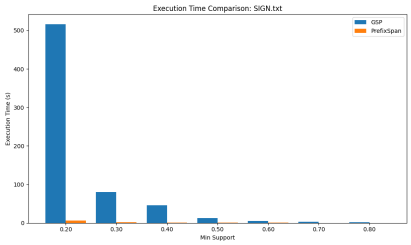
Min_sup	Metric	GSP	PrefixSpan
0.22	Runtime (s)	4.72	0.26
	Memory (MB)	1.38	20.13
	Frequent Patterns	2	2
0.2	Runtime (s)	9.44	0.34
	Memory (MB)	2.01	20.17
	Frequent Patterns	6	6
0.15	Runtime (s)	13.13	0.41
	Memory (MB)	2.04	20.21
	Frequent Patterns	9	9
0.1	Runtime (s)	71.10	0.53
	Memory (MB)	2.12	20.36
	Frequent Patterns	23	23
0.08	Runtime (s)	119.77	0.63
	Memory (MB)	2.80	20.37
	Frequent Patterns	32	32
0.05	Runtime (s)	281.77	0.70
	Memory (MB)	3.65	20.49
	Frequent Patterns	49	49

(c) Comparison Table

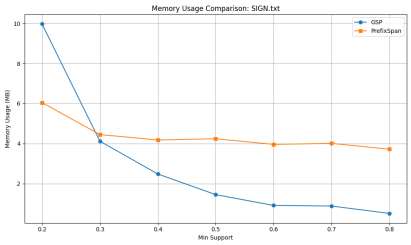
Figure 2: Performance Comparison of GSP vs PrefixSpan (BIKE Dataset)

SIGN

It has total 800 transactions, 267 distinct items and average sequence length is 51.997.



(a) Execution Time



(b) Memory Usage

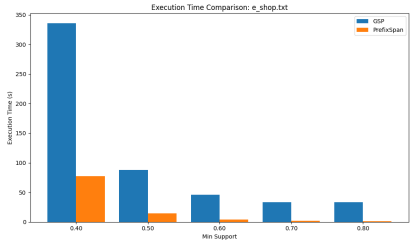
Min_sup	Metric	GSP	PrefixSpan
0.8	Runtime (s)	2.07	0.16
	Memory (MB)	0.51	3.72
	Frequent Patterns	9	9
0.7	Runtime (s)	3.02	0.20
	Memory (MB)	0.88	4.01
	Frequent Patterns	27	27
0.6	Runtime (s)	5.33	0.28
	Memory (MB)	0.91	3.96
	Frequent Patterns	64	64
0.5	Runtime (s)	12.54	0.45
	Memory (MB)	1.45	4.24
	Frequent Patterns	173	173
0.4	Runtime (s)	45.54	0.81
	Memory (MB)	2.48	4.18
	Frequent Patterns	518	518
0.3	Runtime (s)	80.11	1.92
	Memory (MB)	4.12	4.45
	Frequent Patterns	1928	1928
0.2	Runtime (s)	515.59	5.65
	Memory (MB)	9.98	6.05
	Frequent Patterns	9718	9718

(c) Comparison Table

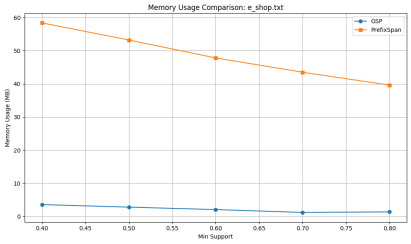
Figure 3: Performance Comparison of GSP vs PrefixSpan (SIGN Dataset)

E-SHOP

It has total 24,026 transactions, 317 distinct items and average sequence length is 61.98.
This data is interesting for algorithm testing because it is sequence of itemsets instead of sequence of items.
As our experiment has some limitations regarding the handling of sequences of itemsets, there are minor discrepancies in the results.



(a) Execution Time



(b) Memory Usage

Min_sup	Metric	GSP	PrefixSpan
0.8	Runtime (s)	33.08	1.39
	Memory (MB)	1.37	39.61
	Frequent Patterns	6	6
0.7	Runtime (s)	33.04	2.23
	Memory (MB)	1.20	43.49
	Frequent Patterns	11	14
0.6	Runtime (s)	46.03	4.20
	Memory (MB)	2.06	47.80
	Frequent Patterns	34	35
0.5	Runtime (s)	87.96	14.82
	Memory (MB)	2.80	53.21
	Frequent Patterns	152	170
0.4	Runtime (s)	335.57	77.17
	Memory (MB)	3.57	58.37
	Frequent Patterns	911	930

(c) Comparison Table

Figure 4: Performance Comparison of GSP vs PrefixSpan (BMSWebView1 Dataset)

5 Conclusion

The results clearly show that the PrefixSpan algorithm discovers frequent sequential patterns faster than GSP. However, it consumes more memory overall, as it retains all intermediate projected databases during execution. Therefore, the choice between PrefixSpan and GSP depends on the specific requirements of the application—whether faster execution or lower memory usage is the priority.

References

[1] R. Srikant and R. Agrawal,
Mining Sequential Patterns: Generalizations and Performance Improvements,
In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT)*, pp. 3–17, 1996.
IBM Almaden Research Center, San Jose, CA.
Available at: `https://eva.fing.edu.uy/pluginfile.php/200637/mod_resource/content/1/Agrawal96_Mining_Sequential_Patterns.pdf`

[2] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu,
PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth,
In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 215–224, Feb. 2001.
doi: `https://doi.org/10.1109/ICDE.2001.914830`