University of Dhaka

Department of Computer Science and Engineering

CSE:4255 Introduction to Data Mining and Warehousing Lab

Lab Report 1: A Comparative Analysis of GSP and PrefixSpan Algorithms

Submitted By:

Name : Ahaj Mahhin Faiak

Roll No : 01

Submitted On :

May 31, 2025

Submitted To :

Dr.Chowdhury Farhan Ahmed, Professor

Md. Mahmudur Rahman, Assistant Professor

1 Introduction

This report compares two popular algorithms for sequantial frequent itemsets mining: **Generalized Sequential Pattern (GSP)** and **PrefixSpan**. Sequential Frequent itemset mining is a fundamental task in data mining, aimed at discovering ordered patterns in transactional datasets. This comparison highlights the differences between the two algorithms in terms of computation time and memory usage across varying support thresholds, using datasets such as BmsWebView1, Sign, Bike and Eshop.

1.1 Objectives

The main objectives of this report are:

- To gain a comprehensive understanding of the operational principles of the GSP and PrefixSpan algorithms.
- To accurately implement both GSP and PrefixSpan algorithms.
- To evaluate and compare the performance of these algorithms in terms of execution time and memory consumption.
- To assess their performance on mining sequential patterns.

2 Theory

2.1 Generalized Sequential Pattern (GSP)

The **Generalized Sequential Pattern (GSP)** algorithm is a classical method for mining frequent sequential patterns in a sequence database. It extends the Apriori algorithm to handle ordered data by using a level-wise, iterative approach. GSP begins by identifying frequent 1-sequences and iteratively generates longer candidate sequences by joining frequent sequences found in the previous iteration. It leverages the Apriori property to prune candidate sequences whose subsequences are not frequent, and it performs multiple database scans to count support at each level.

Although GSP is capable of handling constraints such as time gaps and sliding windows, it can be inefficient on large or dense datasets due to the high computational cost of generating and evaluating a large number of candidates. Nevertheless, GSP is considered a foundational algorithm in sequential pattern mining and serves as a basis for understanding more optimized techniques such as PrefixSpan.

2.2 FP-Growth Algorithm

The **FP-Growth (Frequent Pattern Growth)** algorithm is an efficient method for mining frequent itemsets without generating candidate sets, unlike Apriori-based approaches. It works by compressing the input database into a compact data structure called the *FP-tree* (Frequent Pattern Tree), which retains the itemset association information.

The algorithm performs two main steps: First, it builds the FP-tree by scanning the database twice—once to determine frequent items and their counts, and a second time to construct the tree in a way that groups common prefixes together. Second, it recursively mines the FP-tree using a divide-and-conquer approach, generating frequent itemsets from the conditional pattern bases of each item.

FP-Growth is highly scalable and efficient, especially for large datasets, because it avoids the costly candidate generation step. It is widely used in market basket analysis, web usage mining, and other domains requiring frequent pattern extraction.

3 Implementation Details

3.1 GSP Implementation Details

The GSP algorithm is a classical sequential pattern mining technique used to find frequent sequences in a dataset. The following steps describe the implementation:

- **Reading the Dataset:** The dataset is read from a specially formatted file, where each integer represents an event ID, `-1` denotes the end of an event, and `-2` marks the end of a sequence. These are parsed into a list of sequences, where each sequence is a list of ordered itemsets (sets of event IDs).
- **Generating 1-Sequences:** The algorithm begins by identifying all unique items across sequences to generate candidate 1-sequences. Each item is treated as a single-item sequence, and their support is counted using the `count_support` function. Items whose support meets the minimum support threshold are kept.
- **Candidate Generation:** For $k > 1$, candidate k-sequences are generated from frequent $(k-1)$-sequences. Two sequences are joined if their suffix and prefix (respectively) match after removing one element. The joining logic ensures the generation of meaningful sequential patterns.

- **Pruning Infrequent Subsequences:** Each generated candidate is checked for its direct $(k-1)$-subsequences using the `gen_direct_subsequences` function. If any of its subsequences is not frequent in the previous level, the candidate is pruned using the `prune_cands` function.
- **Support Counting:** The support of each remaining candidate sequence is calculated by checking how many full sequences in the dataset contain the candidate as a subsequence. The `is_subsequence` function determines this using recursive matching of itemsets.
- **Iterative Process:** The process continues iteratively. For each level k, new candidates are generated from the previous level, pruned, and then evaluated. This continues until no more frequent sequences can be found.
- **Output:** The final output is a list of frequent sequences along with their support counts, sorted in descending order of support. This list contains all the sequential patterns that meet the minimum support threshold.

The key functions involved in this implementation include:

- `load_data`: Reads and parses the dataset into sequences of itemsets.
- `count_support`: Counts how many sequences in the dataset contain the given candidate sequence.
- `is_subsequence`: Checks whether a candidate sequence is a subsequence of a given sequence.
- `gen_cands`: Generates new candidate sequences from the previous level of frequent sequences.
- `gen_cands_for_pair`: Merges two frequent sequences to produce a new candidate.
- `gen_direct_subsequences`: Produces all immediate $(k-1)$-subsequences of a candidate sequence.
- `prune_cands`: Removes candidates that contain any infrequent $(k-1)$-subsequence.
- `gsp`: The main function that orchestrates the algorithm by iteratively generating, pruning, and validating candidates, and collecting all frequent sequences.

3.2 PrefixSpan Algorithm Implementation Details

The PrefixSpan algorithm (Prefix-projected Sequential pattern mining) is a pattern growth method used to mine frequent sequential patterns. Below is an overview of the implementation and workflow:

- **Reading the Dataset:** The `load_data` function parses data from a file in SPMF format. Each sequence is represented as a list of itemsets, where:
  - Integers represent event/item IDs.
  - `-1` indicates the end of an itemset (event).
  - `-2` indicates the end of a sequence.
- **Initial Frequent Items:** In `prefix_span`, all frequent 1-item patterns are found by scanning each sequence once. An item is counted once per sequence to avoid overcounting. Only items whose support meets the minimum threshold are retained.
- **Pattern Growth:** For each frequent 1-item pattern:
  - A projected database is created, consisting of the suffixes of sequences after the first occurrence of the item.
  - The `prefix_span_rec` function recursively extends the current pattern with frequent items found in the projected database.
  - Each new pattern is added to the result list if it meets the support threshold.
- **Recursive Extension:** The recursive function `prefix_span_rec` operates by:
  - Scanning the projected database to identify all frequent items (those that occur in the suffixes).
  - Appending each such item to the current pattern to form a new extended pattern.
  - Generating a new projected database for this pattern and recursively repeating the process until the maximum pattern length is reached or no more frequent items remain.
- **Pattern Formatting:** The `format_pattern` function formats discovered patterns for human-readable output using the notation `item_1` $\rightarrow$ `item_2` $\rightarrow$ `....`

- **Execution and Evaluation:** The `run_prefix_span` function manages the overall process:
  - It loads the dataset, computes the absolute support if a relative value is given, runs the PrefixSpan mining procedure, and returns sorted frequent patterns.
  - It also measures execution time and memory usage for performance evaluation.

The main functions involved are:

- `load_data`: Parses SPMF format into sequences of itemsets.
- `prefix_span`: Initiates mining by identifying frequent 1-patterns and invoking recursion.
- `prefix_span_rec`: Recursively grows frequent patterns using projected databases.
- `run_prefix_span`: Executes the mining pipeline and tracks performance.
- `format_pattern`: Provides a human-readable format for discovered patterns.

This implementation of PrefixSpan efficiently mines all frequent sequential patterns up to a given maximum length while ensuring that memory and computation are optimized through pattern projection.

4 Experimental Result

4.1 Bike

From the graphs,we see that for small minimum support values ,the difference in memory-usage and time-consumption is evident. But for larger minimum support, the impact is negligible as it is a sparse graph.

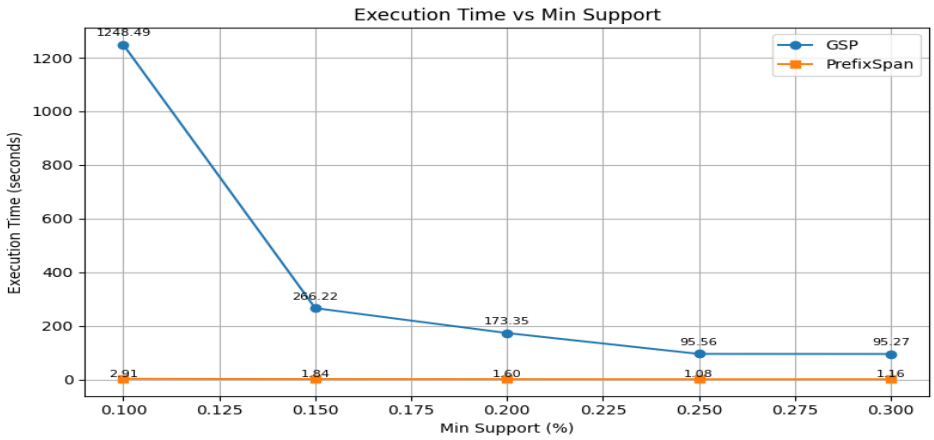

Figure 1: Bike Execution Time

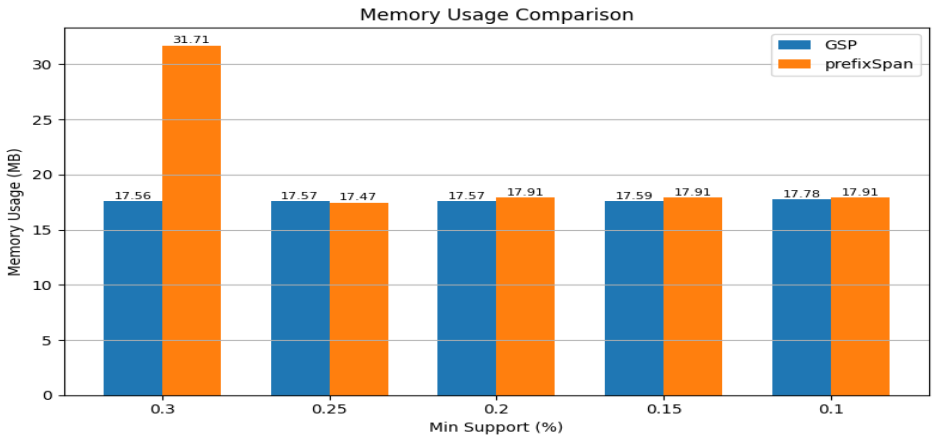

Figure 2: Bike Memory Usage

Performance Comparison: GSP vs PrefixSpan

Dataset: bike.txt

Min_sup	Metric	GSP	PrefixSpan
0.3%	Runtime (s)	95.27	1.16
	Memory (MB)	17.56	31.71
	Frequent Items	0	0
0.25%	Runtime (s)	95.56	1.08
	Memory (MB)	17.57	17.47
	Frequent Items	0	0
0.2%	Runtime (s)	173.35	1.60
	Memory (MB)	17.57	17.91
	Frequent Items	6	6
0.15%	Runtime (s)	266.22	1.84
	Memory (MB)	17.59	17.91
	Frequent Items	9	9
0.1%	Runtime (s)	1248.49	2.91
	Memory (MB)	17.78	17.91
	Frequent Items	23	23

Figure 3: Bike Output

4.2 BmsWebView1

BmsWebView1 had similar memory usage for both algos, the difference in execution time is evident.

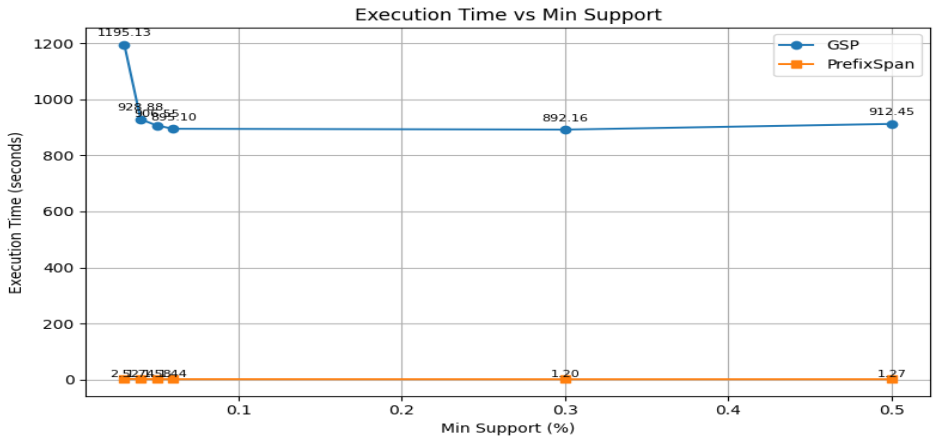

Figure 4: BmsWebView1 Execution Time

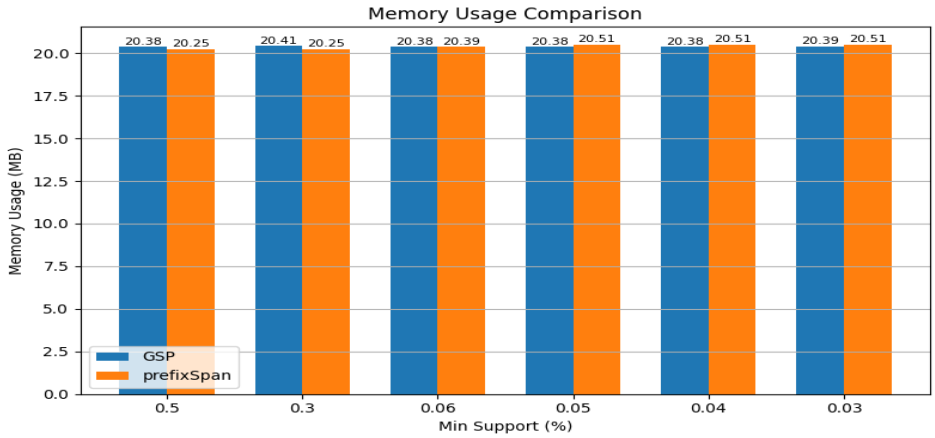

Figure 5: BmsWebView1 Memory Usage

Performance Comparison: GSP vs PrefixSpan
Dataset: BmsWeb1.txt

Min_sup	Metric	GSP	PrefixSpan
0.5%	Runtime (s)	912.45	1.27
	Memory (MB)	20.38	20.25
	Frequent Items	0	0
0.3%	Runtime (s)	892.16	1.20
	Memory (MB)	20.41	20.25
	Frequent Items	0	0
0.06%	Runtime (s)	895.10	1.44
	Memory (MB)	20.38	20.39
	Frequent Items	3	3
0.05%	Runtime (s)	906.55	1.58
	Memory (MB)	20.38	20.51
	Frequent Items	4	4
0.04%	Runtime (s)	928.88	1.74
	Memory (MB)	20.38	20.51
	Frequent Items	5	5
0.03%	Runtime (s)	1195.13	2.52
	Memory (MB)	20.39	20.51
	Frequent Items	11	11

Figure 6: BmsWebView1 Output

4.3 Sign

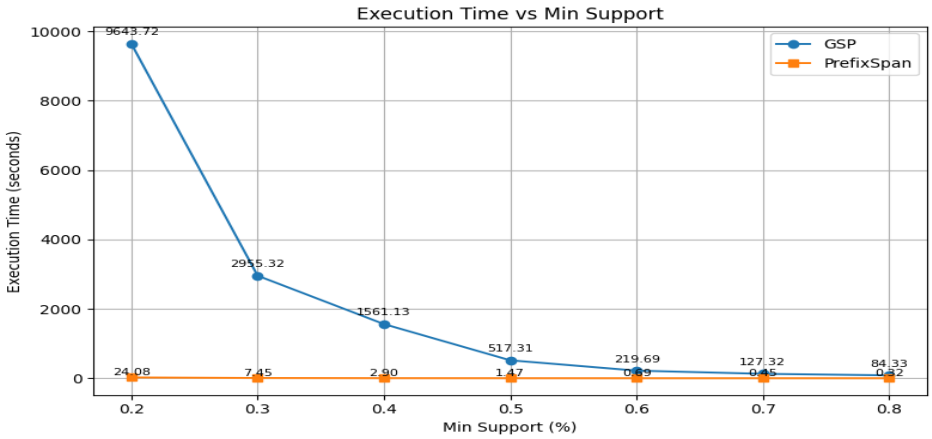

Figure 7: Sign Execution Time

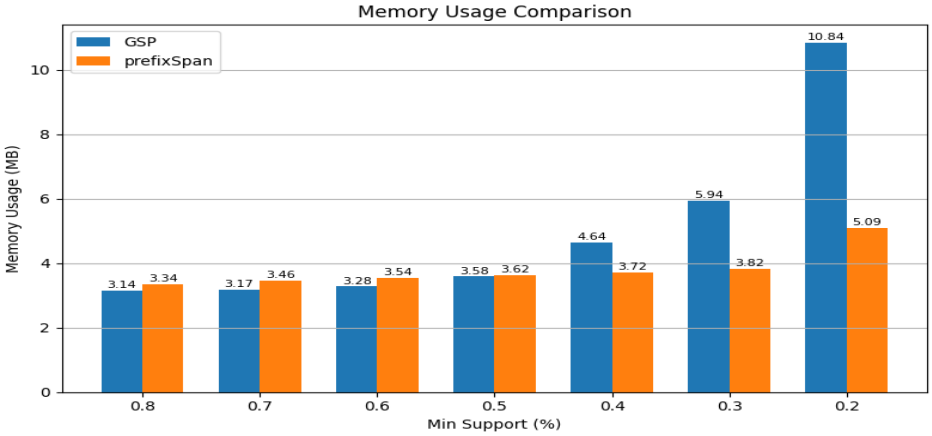

Figure 8: Sign Memory Usage

Performance Comparison: GSP vs PrefixSpan
Dataset: sign.txt

Min_sup	Metric	GSP	PrefixSpan
0.8%	Runtime (s)	84.33	0.32
	Memory (MB)	3.14	3.34
	Frequent Items	9	9
0.7%	Runtime (s)	127.32	0.45
	Memory (MB)	3.17	3.46
	Frequent Items	27	27
0.6%	Runtime (s)	219.69	0.69
	Memory (MB)	3.28	3.54
	Frequent Items	64	64
0.5%	Runtime (s)	517.31	1.47
	Memory (MB)	3.58	3.62
	Frequent Items	173	173
0.4%	Runtime (s)	1561.13	2.90
	Memory (MB)	4.64	3.72
	Frequent Items	518	518
0.3%	Runtime (s)	2955.32	7.45
	Memory (MB)	5.94	3.82
	Frequent Items	1928	1928
0.2%	Runtime (s)	9643.72	24.08
	Memory (MB)	10.84	5.09
	Frequent Items	9718	9718

Figure 9: Sign Output

4.4 E-Shop

E-shop showed strange behaviour where memory usage increased with decreasing

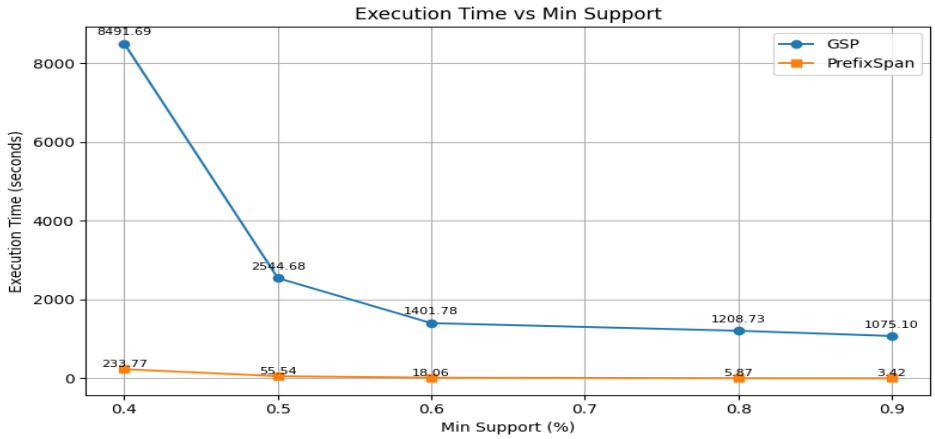

Figure 10: E-Shop Execution Time

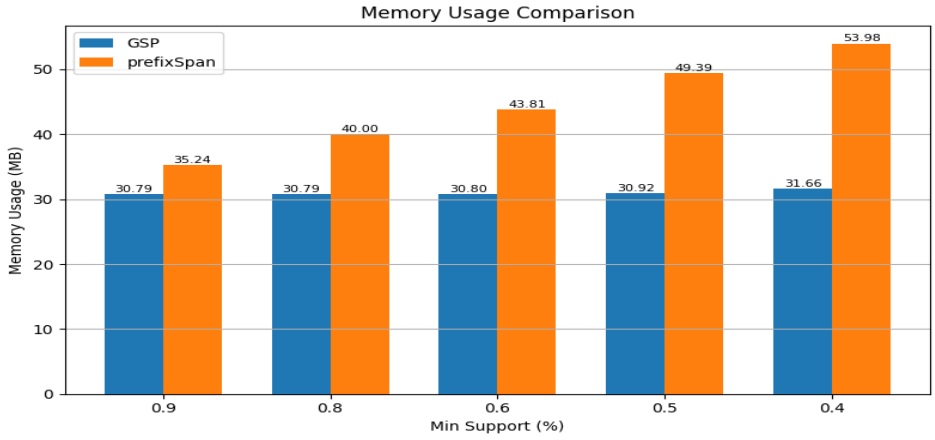

Figure 11: E-Shop Memory Usage

Performance Comparison: GSP vs PrefixSpan

Dataset: eshop.txt

Min_sup	Metric	GSP	PrefixSpan
0.9%	Runtime (s)	1075.10	3.42
	Memory (MB)	30.79	35.24
	Frequent Items	1	1
0.8%	Runtime (s)	1208.73	5.87
	Memory (MB)	30.79	40.00
	Frequent Items	6	6
0.6%	Runtime (s)	1401.78	18.06
	Memory (MB)	30.80	43.81
	Frequent Items	34	35
0.5%	Runtime (s)	2544.68	55.54
	Memory (MB)	30.92	49.39
	Frequent Items	152	170
0.4%	Runtime (s)	8491.69	233.77
	Memory (MB)	31.66	53.98
	Frequent Items	910	930

Figure 12: E-Shop Output

5 Conclusion

The experiment successfully demonstrated the expected outcomes. It was observed that the FP-Growth algorithm generally performed faster than Apriori, although it required more memory. Additionally, the execution time showed a consistent decrease as the minimum support threshold increased.