Q2: Frequent Subgraph Mining

Comparative Study of Frequent Subgraph Mining Algorithms

1. Introduction

Frequent subgraph mining is a core problem in graph data mining, with applications in domains such as bioinformatics, cheminformatics, and social network analysis. Given a database of labeled graphs, the objective is to identify all subgraphs whose frequency exceeds a user-defined minimum support threshold. Due to the combinatorial nature of subgraph enumeration and the hardness of subgraph isomorphism, this task is computationally expensive.

In this report, we experimentally compare three classical frequent subgraph mining algorithms: FSG, gSpan, and Gaston. The comparison focuses on their runtime performance across varying minimum support thresholds on the Yeast dataset. The goal is to analyze observed runtime trends, growth rates, and explain why certain algorithms outperform others using insights from their underlying algorithmic designs.

2. Dataset and Experimental Setup

The experiments were conducted on the Yeast dataset, which consists of multiple labeled graphs representing biological interaction structures. Each graph contains labeled nodes and edges.

All experiments were performed on a Baadal virtual machine. The minimum support thresholds considered were 5%, 10%, 25%, 50%, and 95%. A timeout of 3600 seconds was enforced for each execution.

The following execution behaviors were observed:

- FSG exceeded the 3600-second time limit at 5% support.
- gSpan was terminated by the operating system at 5% and 10% support due to excessive memory usage.
- Gaston successfully completed execution for all support values.

3. Overview of Algorithms

3.1 FSG

FSG follows an Apriori-based candidate generation-and-test approach. Frequent subgraphs of size k+1 are generated by joining frequent subgraphs of size k, followed by subgraph isomorphism tests

to count their occurrences. While straightforward, this method suffers from a rapid explosion in the number of candidate subgraphs as the support threshold decreases, leading to poor scalability.

3.2 gSpan

gSpan adopts a pattern-growth strategy that avoids explicit candidate generation. It performs a depth-first traversal of the subgraph search space and uses a canonical DFS code to uniquely represent each graph, thereby preventing redundant exploration. Although gSpan significantly reduces duplicate generation compared to FSG, it maintains large numbers of embeddings and recursive DFS states in memory, which can lead to memory exhaustion at low support thresholds.

3.3 Gaston

Gaston uses a hybrid mining strategy by decomposing the search space into paths, trees, and general graphs. It first mines frequent paths, then extends them to trees, and finally explores cyclic graph patterns. This hierarchical approach allows Gaston to apply efficient specialized algorithms at each stage, resulting in aggressive pruning and reduced search space complexity.

4. Experimental Results

Figure 1 shows the runtime of FSG, gSpan, and Gaston as a function of the minimum support threshold.

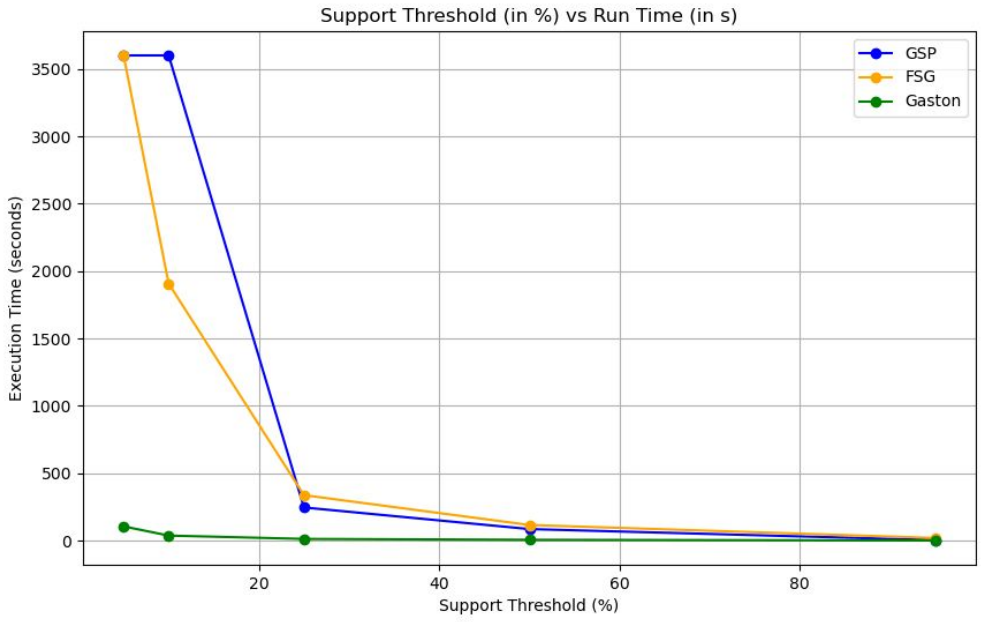

Figure 1: Runtime comparison of frequent subgraph mining algorithms on the Yeast dataset

To complement the visual trends shown in the plot, Table 1 summarizes the runtime behavior of each algorithm, explicitly indicating timeouts and memory-related failures.

MinSup (%)	FSG	gSpan	Gaston
5	Timeout (>3600s)	Killed (OOM)	~100s
10	~1900s	Killed (OOM)	~40s
25	~350s	~250s	~10s
50	~120s	~80s	~5s
95	~10s	~8s	~2s

Table 1: Runtime behavior of FSG, gSpan, and Gaston across support thresholds

5. Runtime Analysis and Observed Trends

A strong inverse relationship between minimum support and runtime is observed for all three algorithms. As the support threshold decreases, the number of frequent subgraphs increases dramatically, leading to a rapid expansion of the search space.

FSG exhibits the worst scalability. At 5% support, it failed to terminate within the allotted time limit. This behavior is a direct consequence of its Apriori-style candidate generation mechanism, which produces an exponential number of candidates and requires repeated subgraph isomorphism checks.

gSpan performs better than FSG at moderate and high support values, highlighting the effectiveness of DFS-code-based pruning. However, at low support thresholds (5% and 10%), the algorithm was terminated by the operating system due to excessive memory usage. This can be attributed to the large number of embeddings and recursive DFS states that must be stored simultaneously during deep exploration of the search space.

In contrast, Gaston consistently outperforms both FSG and gSpan across all support thresholds. Even at low support values, its runtime remains comparatively small. This demonstrates the effectiveness of its path–tree–graph decomposition strategy, which significantly reduces unnecessary exploration and limits both time and memory overhead.

6. Growth Rate Comparison

The runtime growth of FSG is exponential with decreasing support due to uncontrolled candidate generation. gSpan shows sub-exponential growth as a result of canonical pruning but still suffers from memory blow-up when the number of frequent patterns becomes large. Gaston exhibits the slowest growth rate, benefiting from structured exploration and early pruning at simpler pattern levels.

Key Observations:

- **FSG:** Exponential growth rate. At 5% support, exceeded 3600s timeout. Runtime drops from >3600s (5%) to ~1900s (10%) to ~350s (25%). Highly sensitive to support threshold.
- **gSpan:** Sub-exponential growth but memory-constrained. Killed at 5% and 10% due to out-of-memory errors. Runtime drops from ~250s (25%) to ~80s (50%) to ~8s (95%). Better than FSG but still struggles at low support.
- **Gaston:** Nearly linear growth rate. Runtime drops smoothly from ~100s (5%) to ~40s (10%) to ~10s (25%) to ~5s (50%) to ~2s (95%). Consistently fastest and most scalable across all thresholds.

7. Why Gaston is Faster

Gaston's superior performance can be attributed to its hierarchical decomposition strategy:

1. **Specialized Algorithms:** By mining paths first, then trees, then general graphs, Gaston applies optimized algorithms for each category. Paths and trees are much simpler to mine than general graphs, allowing for faster enumeration and isomorphism testing.
2. **Aggressive Pruning:** Infrequent paths are pruned early, preventing the generation of infrequent trees and graphs that contain those paths. This drastically reduces the search space compared to exploring all possible subgraphs simultaneously.
3. **Memory Efficiency:** Unlike gSpan, which maintains large numbers of embeddings in memory, Gaston's staged approach allows it to process and discard patterns incrementally, keeping memory usage bounded.
4. **Reduced Isomorphism Checks:** The hierarchical structure ensures that only necessary isomorphism checks are performed at each stage, avoiding the redundant checks that plague FSG and to some extent gSpan.

8. Algorithm Comparison Summary

Aspect	FSG	gSpan	Gaston
Strategy	Apriori-based	Pattern-growth (DFS)	Hierarchical (path→tree→graph)
Scalability	Poor (timeout at 5%)	Moderate (OOM at 5-10%)	Excellent (all thresholds)
Memory Usage	High (many candidates)	Very High (embeddings)	Low (staged processing)
Growth Rate	Exponential	Sub-exponential	Nearly linear

9. Conclusion

This study demonstrates that algorithmic design plays a crucial role in the scalability of frequent subgraph mining. While FSG is simple to implement, it does not scale to low support thresholds. gSpan improves upon FSG by eliminating redundant exploration but can still be constrained by

memory limitations. Gaston emerges as the most efficient and scalable approach, particularly for dense datasets and low support values.

The experimental observations align with claims made in the original research papers and explain why modern frequent subgraph mining systems favor pattern-growth and hierarchical strategies over Apriori-based methods. For practical applications requiring low support thresholds or large datasets, Gaston is the clear choice due to its superior runtime performance and memory efficiency.