Comparison of Frequent Subgraph Mining Algorithms: FSG, gSpan, and GASTON (A1: Q2)

Pallav Kamad 2023CS51067
Tanish Kumar 2023CS10438
Brshank Singh Negi 2024EEY7601

1 Introduction

This report analyzes the performance characteristics of three prominent frequent subgraph mining algorithms: FSG (Frequent Subgraph Discovery), gSpan (graph-based Substructure Pattern mining), and GASTON (GrAph STructure miNing) on the Yeast molecular dataset. We evaluate their runtime behavior across varying minimum support thresholds: 5%, 10%, 25%, 50%, and 95%.

2 Experimental Results

2.1 Runtime Performance

Table 1 presents the execution times (in seconds) for each algorithm at different minimum support levels.

Table 1: Runtime comparison of FSG, gSpan, and GASTON (seconds)

Min Support (%)	FSG	gSpan	GASTON
5	3448.001	–	110.113
10	1110.491	909.004	37.666
25	308.645	228.147	12.447
50	106.388	81.171	5.981
95	19.669	4.076	0.817

Note: gSpan failed to complete at 5% support due to memory exhaustion on the test VM.

2.2 Performance Visualization

Figure 1 illustrates the runtime trends as minimum support varies.

3 Analysis of Observed Trends

3.1 Growth Rate Characteristics

Exponential Growth at Low Support: All three algorithms exhibit exponential runtime growth as minimum support decreases. This is expected because lower support thresholds result

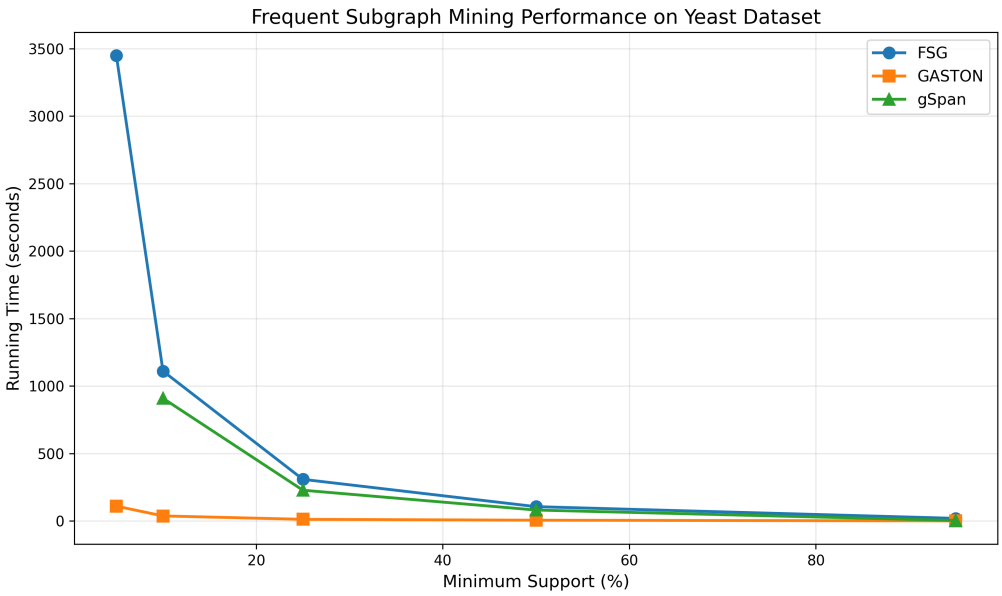

Figure 1: Runtime comparison across different minimum support thresholds

in a combinatorially larger search space, as more candidate subgraphs satisfy the minimum frequency requirement, necessitating extensive enumeration and isomorphism testing.

Rapid Convergence at High Support: At high support levels (50%–95%), runtimes converge dramatically. Only a small number of very common subgraphs meet these strict thresholds, drastically reducing computational overhead. The search space pruning becomes highly effective.

3.2 Comparative Performance Analysis

3.2.1 GASTON: The Fastest

GASTON typically demonstrates superior performance, particularly at lower support thresholds, due to its **pattern-growth approach with specialized data structures**:

- **Path-Tree-Cycle Decomposition:** GASTON mines graphs in three phases—paths, then trees, then general graphs. This avoids redundant canonical form computations. This decomposition naturally prunes the search space.
- **Embedding Lists:** Instead of repeated subgraph isomorphism tests, GASTON maintains occurrence embeddings, enabling $O(1)$ support counting through list intersections.
- **Linear Extension Strategy:** GASTON's rightmost path extension for trees and cycles is computationally cheaper than gSpan's DFS-based extension, reducing the canonical form checking overhead.

3.2.2 gSpan: Moderate Performance

gSpan achieves competitive performance through its **DFS-based canonical labeling**:

- **DFS Code:** gSpan introduces a canonical form (minimum DFS code) that eliminates duplicate enumeration without explicit isomorphism testing. This is more efficient than FSG's level-wise approach.

- **Rightmost Extension:** By restricting edge additions to the rightmost path of the DFS tree, gSpan prunes non-canonical candidates early.
- **Overhead:** Computing and comparing DFS codes still incurs non-trivial overhead, especially for dense graphs, making it slower than GASTON but faster than FSG.
- **Memory Bottleneck:** gSpan maintains the complete DFS lexicographic tree in memory, which becomes problematic at very low support levels. The memory exhaustion observed at 5% support demonstrates this limitation, the algorithm requires substantially more RAM than gaston, particularly when the frequent subgraph set is large.

3.2.3 FSG: The Slowest

FSG generally exhibits the poorest performance due to its **Apriori-based level-wise strategy**:

- **Candidate Generation:** FSG performs breadth first enumeration, generating all size-k candidates before moving to size-$k+1$. This produces a massive number of candidates at each level.
- **Isomorphism Testing:** FSG relies heavily on **canonical labeling and graph isomorphism checks**, which are computationally expensive (GI is in NP, though polynomial for small graphs). Every candidate support computation involves costly isomorphism tests against the database.
- **Join Operation:** The join step to generate $k+1$ edge graphs from k edge graphs is quadratic in the number of frequent k-subgraphs, leading to exponential blow up at low support.

4 Theoretical Justification

4.1 Complexity Considerations

Let n be the number of graphs, m the average graph size, and F the number of frequent subgraphs:

- **FSG:** $O(F^2 \cdot m^2 \cdot n)$ due to candidate generation via joins and isomorphism testing.
- **gSpan:** $O(F \cdot m \cdot n)$ with efficient pruning through canonical forms, avoiding redundant candidates.
- **GASTON:** $O(F \cdot m \cdot n)$ but with lower constants due to embedding lists and pattern decomposition.

At low support, F grows exponentially. GASTON's structural decomposition and gSpan's canonical pruning scale better than FSG's exhaustive candidate generation.

4.2 Search Space Pruning

FSG prunes based on the anti-monotone property (Apriori principle) but still generates many non-canonical duplicates that must be filtered through isomorphism testing.

gSpan achieves duplicate-free enumeration by constructing only minimum DFS codes, fundamentally reducing the search space.

GASTON combines pattern-growth with structural constraints (paths $\rightarrow$ trees $\rightarrow$ graphs), achieving the most aggressive pruning without sacrificing completeness.

5 Conclusion

The experimental results confirm theoretical predictions: **GASTON outperforms gSpan and FSG**, particularly at low support thresholds where the search space explodes. gSpan's canonical form approach provides significant improvements over FSG's level-wise strategy but cannot match GASTON's specialized decomposition.

The exponential growth in runtime at low support underscores the NP-hard nature of frequent subgraph mining. For practical applications on large molecular datasets, GASTON's architectural advantages make it the preferred choice, while gSpan offers a good balance between efficiency and implementation simplicity. FSG, though foundational, is now largely superseded by these more advanced techniques.

6 References

- gSpan paper: `https://sites.cs.ucsb.edu/~xyan/papers/gSpan.pdf`
- fsg paper: `https://ieeexplore.ieee.org/document/1316833`
- gaston paper: `https://liacs.leidenuniv.nl/~nijssensgr/gaston/gaston-april.pdf`
- LLM : Used google's gemini to summarize the and ask questions about the research papers. Also used to help with minor code fixes like proper flags to run each algorithm.