COL 761 HW1 Report Question 2 Analysis

February, 2025

Team members	Entry Number
Hasit Nanda	2024VST9015
Ghosal Subhojit	2022MT61976
Arnab Goyal	2022MT61963

This analysis presents an empirical comparison of **gSpan**, **FSG**, and **Gaston** algorithms' performance across different support thresholds and explains the observed trends based on the algorithm execution.

1 Observed Performance Results

The following table summarizes the execution times for different support thresholds:

Support Threshold (%)	FSG Time (s)	gSpan Time (s)	Gaston Time (s)
95%	25.88	6.71	1.38
50%	142.30	117.20	9.20
25%	410.19	331.09	20.25
10%	1448.07	1649.10	61.10
5%	TLE	MLE	172.29

Table 1: Execution times of FSG, gSpan, and Gaston at different support thresholds.

2 Algorithm Explanations

To better understand the observed performance trends, we first provide an overview of how each algorithm operates.

2.1 gSpan (Graph-based Substructure Pattern Mining)

gSpan eliminates candidate generation by using a **depth-first search (DFS) code** to lexicographically order subgraphs. Instead of explicitly checking subgraph isomorphism, it prunes redundant patterns early using a **canonical labeling** approach. This enables efficient subgraph mining without the overhead of explicit isomorphism tests.

Key Features:

- Uses DFS lexicographic ordering to reduce redundant candidate generation.

- Efficiently prunes the search space by maintaining a canonical order.
- Works well on **sparse graphs** but can struggle with memory usage on dense datasets. This is because sparse graphs have fewer possible subgraphs, reducing the DFS traversal computations. Dense graphs, on the other hand, have many interconnected subgraphs, and gSpan keeps all DFS embeddings in memory, leading to memory exhaustion.

2.2 FSG (Frequent Subgraph Mining)

FSG is an **Apriori-based approach** that generates candidate subgraphs and prunes them based on frequency. It follows a level-wise search strategy, expanding smaller frequent subgraphs into larger ones. However, it requires **subgraph isomorphism checking** at each step, which is computationally expensive.

Key Features:

- Uses a level-wise approach similar to Apriori for subgraph generation.
- Requires frequent subgraph isomorphism tests, making it slow at low support.
- Performs well on **small dense graphs** but suffers from candidate explosion as support decreases. This is because dense graphs contain many overlapping substructures, making candidate generation useful. In sparse graphs, fewer frequent patterns exist, so FSG wastes time generating unnecessary candidates.

2.3 Gaston (Graph-Based Substructure Mining)

Gaston optimizes subgraph mining by **separately processing paths, trees, and graphs**. It first mines frequent **paths**, then extends to **trees**, and finally expands to **general graphs**. By leveraging this structured approach, Gaston reduces unnecessary computations and improves runtime efficiency.

Key Features:

- Separates mining of paths, trees, and graphs to optimize pattern growth.
- Uses **pattern-based partitioning** to efficiently reduce redundant searches.

3 Algorithmic Performance Trends

To understand the performance differences, we analyze the runtime characteristics of each algorithm across different support levels.

3.1 High Support Thresholds (95% and 50%)

Key Observations:

- Gaston is the fastest across all methods at high support thresholds due to its optimized pattern growth approach.
- gSpan performs significantly better than FSG at 95% support because DFS traversal is well-suited for small frequent patterns. However, at 50%, gSpan slows down due to increasing memory requirements.
- FSG is slower due to its Apriori-style candidate generation and the need for subgraph isomorphism checking.

3.2 Medium Support Thresholds (25% and 10%)

Key Observations:

- Gaston remains the fastest, allowing efficient mining of moderately frequent subgraphs.
- gSpan outperforms FSG due to its ability to prune unpromising search paths early, reducing redundant computations.
- FSG struggles due to candidate explosion, as it generates a large number of subgraph candidates, many of which require expensive isomorphism tests.

3.3 Low Support Threshold (5%)

Key Observations:

- FSG fails due to excessive computation time (TLE), as the number of candidate subgraphs becomes too large to process.
- gSpan fails due to memory overflow (MLE), as storing and processing numerous DFS embeddings consumes too much memory.
- Gaston, while slower than at higher support levels, remains the only viable method at 5% support due to its ability to prune redundant computations.

4 Why Gaston Works Best Overall

Key Feature: Gaston optimizes subgraph mining by processing paths, trees, and full graphs separately.

Unlike gSpan and FSG, which apply a uniform approach to mining subgraphs, **Gaston** introduces a structured mining strategy that differentiates between simple and complex subgraphs.

Optimized Search Strategy Gaston divides subgraph mining into three phases:

1. **Mine paths first** – Since paths are the simplest structures, mining them first allows quick discovery of frequent substructures.
2. **Extend to trees** – Once frequent paths are identified, they are extended into tree-like subgraphs.
3. **Expand to full graphs** – Finally, Gaston discovers more complex graph structures by expanding from frequent trees.

Why this is better:

- Avoids premature generation of complex subgraphs (unlike FSG, which immediately considers multi-node structures).
- Optimally reduces the search space by leveraging simpler patterns first.

Compared to gSpan:

- **Lower Memory Overhead** – Instead of maintaining a deep DFS search tree like gSpan, Gaston uses incremental substructure growth.
- **Faster Processing of Trees and Paths** – Many real-world datasets contain frequent tree-like structures. Gaston mines these first, making it faster than gSpan.

Compared to FSG:

- **Less Candidate Explosion** – While FSG generates all possible subgraphs at each step (leading to exponential computation), Gaston reduces this by applying pattern growth, limiting unnecessary candidates.
- **More Scalable** – Instead of frequent subgraph isomorphism checking at every step (which slows FSG down), Gaston selectively grows subgraphs, reducing unnecessary computations.

5 Conclusion

Factor	FSG	gSpan	Gaston
Execution Time	Slow at low support	Fails due to memory	Fastest overall
Scalability	Candidate explosion	Memory bottleneck	Best at moderate support
Best for	Small dense graphs	Large sparse graphs	Moderate-sized datasets

Table 2: Summary comparison of FSG, gSpan, and Gaston.

Key Takeaways:

- FSG is feasible at high support but slows considerably at low support due to candidate explosion.
- gSpan is efficient for sparse graphs but fails at low support due to memory exhaustion.
- Gaston is the extremely fast compared to FSG and gSpan.