Analysis of Frequent Subgraph Mining Algorithms on Yeast Dataset

Methodology & Implementation

To compare the performance of gSpan, FSG, and Gaston, we performed the following preprocessing and experimental steps on the Yeast dataset:

- **Data Preprocessing:** The original dataset (167.txt_graph) contained string labels (e.g., "C", "O"). Since the standard binaries require integer inputs, we developed a Python script to:
    1. Parse the graph data and map unique string labels to integers (e.g., C → 0, O → 1).
    2. Format the data specifically for gSpan, FSG, and Gaston.
    3. We also explicitly appended a newline character (\n) to end-of-file markers to prevent input stream freezing in the C++ binaries.

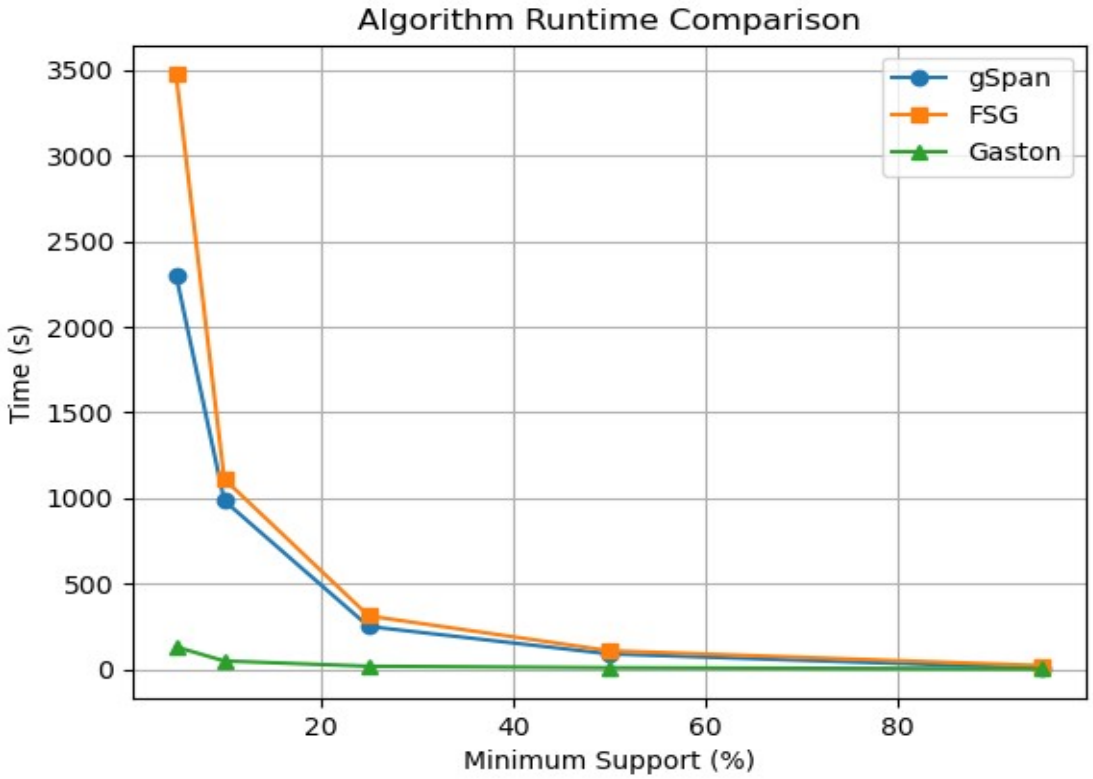
- **Experimental Setup:**
    1. **Platform:** The algorithms were executed on a Baadal VM (Ubuntu 22.04 Server, 4 vCPUs, 8GB RAM, 80GB HDD).
    2. **Parameters:** Execution time was measured for Minimum Support thresholds of **95%, 50%, 25%, 10%, and 5%**.
    3. **Metric:** Total Wall Clock time (in seconds).

Observations

Based on the generated runtime plot, we observed the following:

- **Exponential Growth:** All algorithms showed a drastic increase in runtime as support dropped from 10% to 5%. This reflects the exponential growth in the number of frequent subgraphs.
- **Algorithm Ranking (Fastest to Slowest):**
    - **Gaston (Fastest):** Gaston remained incredibly efficient, with the runtime line staying near the bottom (under 200s) even at 5% support.

- **gSpan (Intermediate):** gSpan performed well until 10%, but spiked significantly at 5% support, reaching approximately 2,300 seconds.
- **FSG (Slowest):** FSG struggled the most with the dense dataset at low support. It showed the steepest curve, peaking at approximately 3,500 seconds for the 5% threshold.



Analysis of Results

The performance differences can be attributed to the algorithmic strategies:

- **Why Gaston was Fastest:** Gaston uses a "Hybrid" approach (Path → Tree → Cyclic). Since molecular data (like Yeast) consists mostly of trees and paths, Gaston mines these phases efficiently in polynomial time and only performs expensive isomorphism checks for cycles.
- **Why FSG was Slowest:** FSG is a Breadth-First Search (BFS) algorithm that generates candidates level-by-level. At 5% support, the number of frequent subgraphs is massive. FSG suffers from a bottleneck in **candidate generation**, where it must join smaller graphs to create larger candidates and then verify them, leading to high memory and CPU usage.

- **Why gSpan Spiked:** gSpan is a Depth-First Search (DFS) algorithm. It avoids candidate generation (saving memory compared to FSG) but suffers from the overhead of **subgraph isomorphism testing** during its recursive growth. At low support, the search depth increases, making these checks computationally expensive.

Conclusion

The experiment confirms that **Gaston** is the superior choice for the Yeast dataset, leveraging its hybrid strategy to outperform the pure DFS (gSpan) and BFS (FSG) approaches. While FSG and gSpan are effective at higher supports, they face significant scaling challenges at low support thresholds (5%) due to candidate generation and recursion overheads, respectively.