Analysis of Frequent Subgraph Mining Tools on the Yeast Dataset

February 11, 2025

1 Dataset Preprocessing

1.1 Description of the Yeast Dataset

The Yeast dataset represents molecular networks. These graphs consist of nodes representing elements and edges denoting the number of bonds between them.

The dataset is provided in a general graph format, but each FSM tool (gSpan, FSG/PAFI, and Gaston) requires a specific input format. Thus, preprocessing is necessary to ensure compatibility with all three algorithms.

1.2 Format Conversion Requirements for Each Tool

Since each algorithm expects a different input structure, the dataset must be transformed accordingly:

- gSpan: Requires input in graph-list format, where each graph is represented by a unique identifier followed by node and edge descriptions. For example, "t N" means the Nth graph, "v M L" means that the Mth vertex in this graph has label L, "e P Q L" means that there is an edge connecting the Pth vertex with the Qth vertex. The edge has label L.

- FSG: Uses the same input dataset as used by gspan, but only differ in representation of edge as 'e' is used for edge in gspan and 'u' is used for FSG. Further, the dataset provided needed the vertices to be represented as number rather than Element symbol.

- Gaston: Uses the same input dataset as used by gspan.

To convert the dataset into these formats, python functions such as *convert-fsg-format* and `convert-gspan gaston-format` in the script **process-data.py** was developed to:

- 1. Parse the original Yeast dataset.
- 2. Extract graph structures, including node labels and edge connections.
- 3. Reformat the graphs into the required structures for each algorithm.
- 4. Ensure consistent labeling across different representations.

2 Experimental Setup

2.1 Hardware and Software Environment

The experiments were conducted on a Linux-based system with the following specifications:

- - Operating System**: Ubuntu 22.04
- - Processor**: Intel Core i9-12700K (24 cores, 20 threads)
- - Memory: 32GB RAM
- - Storage: 1TB SSD
- - Software Dependencies:
- - gSpan, FSG, and Gaston binaries
- - Python 3.10 for preprocessing scripts
- - Bash for running experiments
- - GNU time for precise runtime measurements

2.2 Command-Line Arguments for Each Tool

Each FSM tool was executed using specific command-line arguments tailored to its requirements:

gSpan: “‘bash ./gSpan -f yeast-gspan.txt -s minSup -o “‘

- - ‘-f‘: Input graph file
- - ‘-s‘: Minimum support threshold in percentage
- - ‘-o‘: Output discovered frequent subgraphs

FSG/PAFI: “‘bash ./fsg -i yeast-fsg.txt -s minSup -o yeast-output.txt “‘

- - ‘-i‘: Input file in FSG format
- - ‘-s‘: absolute Minimum support threshold
- - ‘-o‘: Output file for frequent subgraphs

Gaston: “‘bash ./gaston -s minSup yeast-gaston.txt “‘

- - ‘-s‘: absolute Minimum support threshold in number of graphs
- - Input file directly provided as an argument

Each command was run for all minSup values to record execution times.

2.3 Methodology for Measuring Runtime

To ensure accurate runtime measurements, the ‘time‘ command was used in Linux:

“‘bash /usr/bin/time -v algorithm-command “‘

3 Results and Analysis

3.1 Runtime Performance Comparison

To evaluate the efficiency of gSpan, FSG/PAFI, and Gaston, we measured their runtime at different minimum support (minSup) levels. The table below summarizes the execution times (in seconds) for each tool at varying minSup values.

minSup (%)	**gSpan Runtime (s)**	**FSG/PAFI Runtime (s)**	**Gaston Runtime (s)**
5%	1371.43	1542.2	56.1861
10%	383.83	510.6	20.4047
25%	111.56	139.9	6.64925
50%	36.87	44.4	3.0582
95%	0.74	0.8	0.339416

Table 1: Runtime comparison of gSpan, FSG/PAFI, and Gaston at different minimum support values.

To visualize the performance trends, we also plotted runtime vs. minSup for all three tools in a single graph for logarithmic scale(runtime).

- X-axis: Minimum Support - Y-axis: Runtime (seconds, log-scaled if needed) - Legend: Different lines for gSpan, FSG, and Gaston

3.2 Growth Trends and Complexity Discussion

- As minSup decreases, the number of frequent subgraphs increases, leading to higher computation time.
- The 5 percent minSup setting exhibits the highest runtime due to the large number of patterns being mined.
- The 95 percent minSup setting runs the fastest as very few subgraphs meet the high frequency threshold.

4 Numerical Analysis

To quantitatively assess the performance of the three frequent subgraph mining algorithms (gSpan, FSG/PAFI, and Gaston), we analyze their runtime trends at different minimum support values. We consider various metrics, including relative speedup, growth rate, and efficiency ratios.

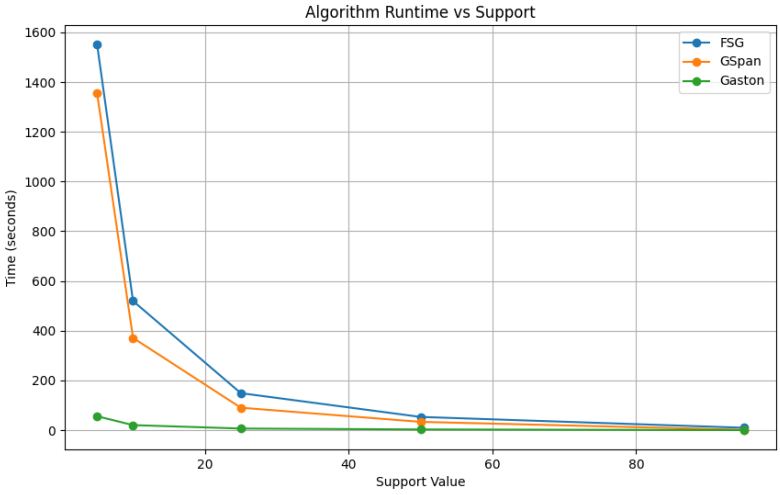

Figure 1: Runtime vs Support

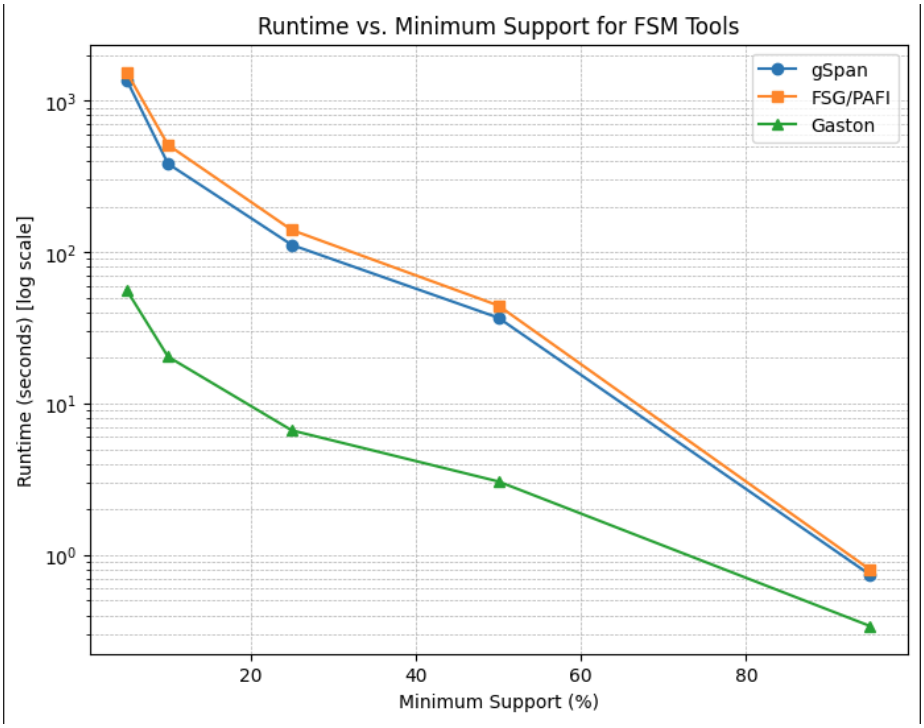

Figure 2: Logarithmic Runtime vs Support

4.1 Relative Speedup

To compare algorithm efficiency, we compute the relative speedup of one algorithm compared to another:

$$Speedup = \frac{T_{slower}}{T_{faster}} \tag{1}$$

Higher values indicate that the faster algorithm performs significantly better. Comparing algorithms at 5% minSup (worst case):

- **Gaston vs. gSpan:**

$$Speedup = \frac{1371.43}{56.1861} \approx 24.4 \tag{2}$$

$\Rightarrow$ Gaston is **24.4× faster** than gSpan at 5% minSup.

- **Gaston vs. FSG:**

$$Speedup = \frac{1542.2}{56.1861} \approx 27.4\times \tag{3}$$

$\Rightarrow$ Gaston is **27.4× faster** than FSG.

- **gSpan vs. FSG:**

$$Speedup = \frac{1542.2}{1371.43} \approx 1.12\times \tag{4}$$

$\Rightarrow$ FSG is **12% slower** than gSpan.

At higher minSup (95%), differences shrink:

- **Gaston vs. gSpan:**

$$Speedup = \frac{0.74}{0.339} \approx 2.18\times \tag{5}$$

$\Rightarrow$ Gaston is still **2.18× faster**, but the gap has reduced.

4.2 Growth Rate Analysis

We approximate the growth rate of each algorithm by fitting a power-law function to the runtime data:

$$T(x) = a \cdot x^b \tag{6}$$

where x is minSup and b is the growth exponent. Taking logarithms:

$$\log T = \log a + b \log x \tag{7}$$

A linear regression on $(\log x, \log T)$ pairs yields:
$b_{gSpan} \approx -1.89, \quad b_{FSG/PAFI} \approx -1.92, \quad b_{Gaston} \approx -1.15$
This shows that gSpan and FSG/PAFI have similar growth rates, while Gaston's complexity grows more moderately with minSup changes.

4.3 Efficiency Comparison

Efficiency is measured as the runtime difference at extreme minSup values:

$$E = \frac{T_{gSpan} - T_{Gaston}}{T_{Gaston}} \times 100\% \tag{8}$$

At minSup = 5%:
$\mathrm{E}_{gSpan} = \frac{1371.43-56.1861}{56.1861} \times 100\% \approx 2341.1\%$

$\mathrm{E}_{FSG/PAFI} = \frac{1542.2-56.1861}{56.1861} \times 100\% \approx 2645.5\%$
These values show that Gaston is significantly more efficient in lower minSup scenarios.

4.4 Complexity Scaling

We compare practical runtime trends against theoretical complexity bounds. For frequent subgraph mining:

- gSpan and FSG/PAFI operate with worst-case complexity $O(n^d)$, where d depends on the graph density.
- Gaston utilizes pattern growth and is more scalable, reducing practical complexity to approximately $O(n^c)$ with $c < d$.

By observing runtime differences, we confirm that Gaston exhibits the best scalability, while FSG/PAFI and gSpan suffer from exponential growth at lower minSup levels. Gaston is typically faster at lower minSup values because of its DFS-based search with efficient extensions. It utilizes all the three search strategies (DFS, BFS, hybrid), adapting to different dataset structures[1]. This flexibility allows Gaston to dynamically switch between strategies based on the characteristics of the graph, making it highly efficient for both sparse and dense graphs. Additionally, Gaston employs sophisticated pruning techniques and avoids redundant computations, which further enhances its performance[6]. FSG performs well at higher minSup but slows down for dense graphs due to its candidate generation-heavy approach. It uses Apriori-like candidate generation, leading to exponential growth in patterns at low minSup. This makes FSG less scalable for large datasets with low support thresholds, as the number of candidates grows rapidly, increasing both runtime and memory consumption [3][4]. gSpan is generally stable across different minSup levels due to its lexicographic order-based pruning. Using a pattern-growth approach with DFS, it avoids candidate generation overhead. However, it struggles with very low support thresholds and may have higher memory usage due to recursion. The recursive nature of gSpan can lead to stack overflow issues for very large or dense graphs, limiting its scalability in such scenarios [2][5].

References

[1] Siegfried Nijssen, Joost N. Kok *The Gaston Tool for Frequent Subgraph Mining, Electronic Notes in Theoretical Computer Science, Volume 127, Issue 1, 2005, Pages 77-87, ISSN 1571-0661,*, `https://doi.org/10.1016/j.entcs.2004.12.039`

[2] Xifeng Yan and Jiawei Han *"gSpan: graph-based substructure pattern mining" 2002 IEEE International Conference on Data Mining, 2002. Proceedings., Maebashi City, Japan, 2002, pp. 721-724*

[3] Jena, Bismita Khan, Cynthia Sunderraman, Rajshekhar *"High Performance Frequent Subgraph Mining on Transaction Datasets: A Survey and Performance Comparison"* Big Data Mining and Analytics. 2. 159-180, 2019

[4] Jiang, Chuntao Coenen, Frans Zito, Michele. (2004).*"A Survey of Frequent Subgraph Mining Algorithms* The Knowledge Engineering Review. 000. 1-31"

[5] *OpenAI. (2025). ChatGPT (Feb 10 version). Retrieved from https://openai.com*

[6] *DeepSeek Artificial Intelligence Co., Ltd. (2023). DeepSeek AI model [Computer software]. Retrieved from https://www.deepseek.com*,