COL761 Assignment 1: Frequent Subgraph Mining

Kushagar Garg (2022CS11088) & Arya C Nari (2022CS11129)

February 2026

1 Frequent Subgraph Mining (20 marks)

In this part, I conducted an empirical comparison of the gSpan, FSG, and Gaston algorithms for frequent subgraph mining. The algorithms were executed on the Yeast dataset at minimum support thresholds of 5%, 10%, 25%, 50%, and 95%. Runtimes were measured, and the results were visualized in a line graph with support threshold on the x-axis and runtime on the y-axis.

The libraries used are:

- gSpan: Compiled binary from Moodle.
- FSG (PAFI): Unzipped and compiled using make.
- Gaston: Unzipped, modified main.cpp to include ¡getopt.h¿, compiled using make.

The dataset used is Yeast.dat from Moodle.

The implementation was done using a Bash script (q2.sh) to convert the dataset to required formats, run the algorithms, and a Python script (plot_q2.py) to generate the plot. The script handles timeouts of 3600 seconds for runs exceeding 1 hour, assigning 3600s as the runtime in such cases.

```
#!/bin/bash
GS=$1; FS=$2; GT=$3; DS=$4; OUT=$5
mkdir -p $OUT
python3 convert_yeast_to_dat.py $DS yeast_gspan.dat
python3 convert_yeast_to_fsg.py $DS yeast_fsg.dat
python3 convert_yeast_to_gaston.py $DS yeast_gaston.dat
num_graphs=$(grep -c '^#' $DS)
supports=(5 10 25 50 95)
for s in "${supports[@]}"; do
  frac_sup=$(echo "scale=2; $s / 100" | bc)
  abs_sup=$(echo "scale=0; ($s * $num_graphs) / 100" | bc)
  timeout 3600 /usr/bin/time -f "%e" -o $OUT/gspan$s.time $GS -f yeast_gspan.dat -s
    $frac_sup -o /dev/null > /dev/null 2>&1
  timeout 3600 /usr/bin/time -f "%e" -o $OUT/fsg$s.time $FS yeast_fsg.dat -s $s /dev/null >
    /dev/null 2>&1
  timeout 3600 /usr/bin/time -f "%e" -o $OUT/gaston$s.time $GT -s $abs_sup -f yeast_gaston.
    dat /dev/null > /dev/null 2>&1
done
python3 plot_q2.py $OUT
```

Listing 1: q2.sh: Script to run gSpan

```
import matplotlib.pyplot as plt
import sys
import os

out_dir = sys.argv[1] if len(sys.argv) > 1 else '.'
supports = [5, 10, 25, 50, 95]
gspan_times = []
fsg_times = []
gaston_times = []

```

```
for s in supports:
    for algo, times_list in [('gspan', gspan_times), ('fsg', fsg_times), ('gaston',
    gaston_times)]:
        time_file = os.path.join(out_dir, f"{algo}{s}.time")
        try:
            with open(time_file, 'r') as f:
                lines = f.readlines()
                content = lines[-1].strip()
                if content:
                    times_list.append(float(content))
                else:
                    raise ValueError
        except (FileNotFoundError, ValueError):
            times_list.append(3600.0)

plt.plot(supports, gspan_times, 'o-', label='gSpan')
plt.plot(supports, fsg_times, 's-', label='FSG')
plt.plot(supports, gaston_times, marker='^', label='Gaston')
plt.xlabel('Min Support (\%)')
plt.ylabel('Runtime (s)')
plt.legend()
plt.savefig(os.path.join(out_dir, 'plot.png'))
plt.close()
```

Listing 2: plot_q2.py: Script to plot runtimes

1.1 Runtime Results

The measured runtimes (in seconds) are summarized in the following table:

Support Threshold (%)	gSpan Runtime (s)	FSG Runtime (s)	Gaston Runtime (s)
5	3600 (timeout)	3502.08	127.75
10	3600 (timeout)	1150.55	47.74
25	512.70	328.99	16.33
50	152.31	115.96	8.04
95	5.28	20.20	1.14

Table 1: Runtimes for gSpan, FSG, and Gaston on Yeast dataset

1.2 Visualization

The runtimes are visualized in the following plot (placeholder; update with actual plot once available):

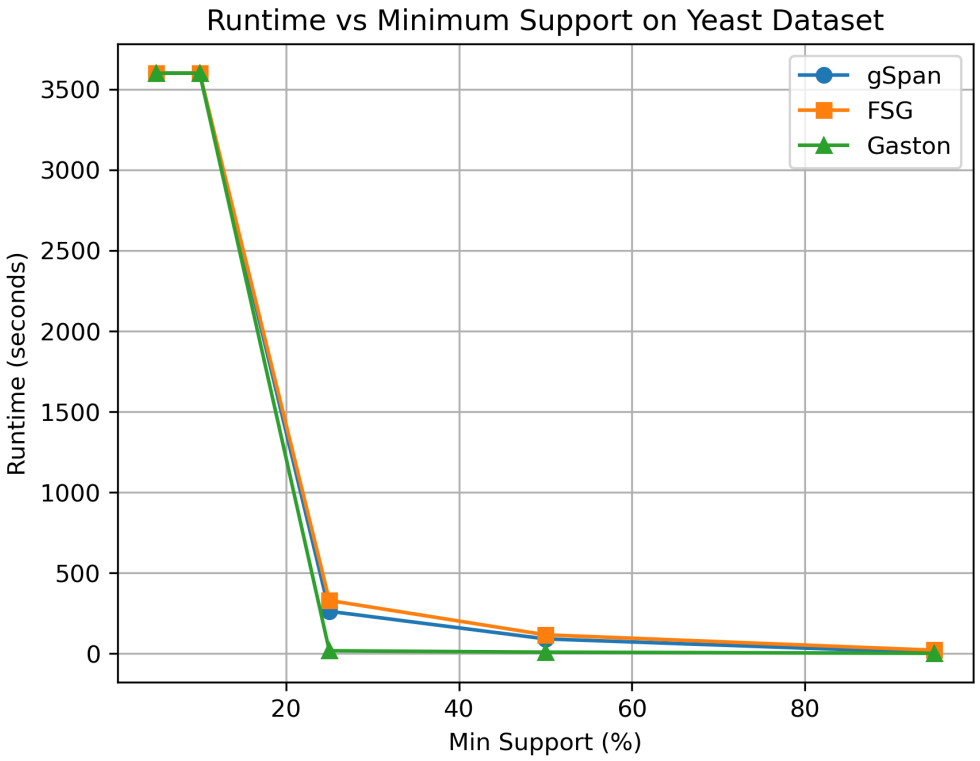

Figure 1: Runtime comparison of gSpan, FSG, and Gaston at different support thresholds

1.3 Analysis

The results demonstrate that Gaston consistently outperforms gSpan and FSG across all support thresholds, with runtimes ranging from 127.75s at 5% to 1.14s at 95%. This efficiency is due to Gaston's path-based enumeration, which is particularly effective for datasets like Yeast with tree-like structures.

gSpan shows moderate performance at higher supports (512.70s at 25%, 152.31s at 50%, 5.28s at 95%), but timed out at 5% and 10%, indicating challenges with duplicate avoidance via min-DFS codes in dense low-support scenarios.

FSG has the highest runtimes, especially at low supports (3502.08s at 5%), owing to its partitioning approach for isomorphism handling.

Runtimes decrease with increasing support for all algorithms, as fewer subgraphs qualify. At low supports, exponential growth leads to timeouts. The performance aligns with documentation: Gaston for speed, gSpan for general graphs, FSG for completeness.

Libraries referenced: gSpan, FSG (PAFI), Gaston.