COL761/COL7361/AIL7026 Data Mining

Assignment 1, Semester-II, 2025-26

Submission deadline: 11:59 PM, 02-Feb-2026

Instructions

‼️ **Add `https://github.com/2023col761` as a collaborator to your github repository**.

- This assignment consists of the following parts:
  - Frequent itemset mining (35 marks)
  - Frequent subgraph mining (20 marks)
  - Graph indexing (45 marks, competitive)
- Your latest git push will be counted as your submission time.
- Submission must be on `https://moodlenew.iitd.ac.in/login/index.php`, under COL761.
- Upload your assignment under `A1` directory, with the directory structure as follows:

```
A1/
  |-- q1/
  |-- q2/
  |-- q3/
```

- The directory structures of `q1`, `q2`, `q3` are indicated under their problem statements.
- Do not submit data files provided by us.

🚫 Anti-Plagiarism Policy

Any detected attempts at plagiarism from parallel/past submissions will result in an **F** grade in the course. **Add references to any libraries used in your report.**

1 Frequent itemset mining (35 marks)

1.1 Task 1: Comparision of `Apriori` and `FP-Tree` algorithms

[5 (implementation) + 8 (report) marks]

Conduct an empirical comparison of the `Apriori` and `FP-Tree` algorithms for frequent itemset mining. Utilizing provided libraries, analyze their efficiency on the given dataset. Measure the runtime of both algorithms at five different support thresholds: 5%, 10%, 25%, 50%, 90%. Subsequently, visualize the results by plotting line graphs on a single plot with the support threshold on the x-axis and runtime on the y-axis. Finally, provide a comprehensive analysis of your observations in a written report, comparing the performance characteristics of the algorithms.

- **Libraries:** Refer to the following documentation for the runtime arguments and input formats. Compile the libraries from source code (e.g., using the `make` command).
  - `Apriori`:
    - Source code: `https://borgelt.net/src/apriori.zip`
    - Documentation: `https://borgelt.net/doc/apriori/apriori.html`
  - `FP-Tree`:
    - Source code: `https://borgelt.net/src/fpgrowth.zip`
    - Documentation: `https://borgelt.net/doc/fpgrowth/fpgrowth.html`
- **Dataset:**
  - Dataset: `http://fimi.uantwerpen.be/data/webdocs.dat.gz`
  - Description: `http://fimi.uantwerpen.be/data/webdocs.pdf`

1.2 Task 2: Dataset creation for the given runtime plot [22 marks]

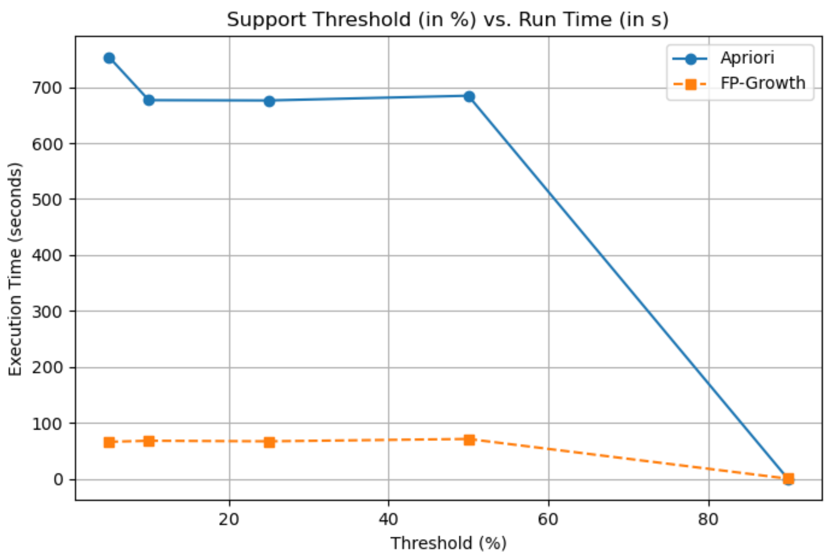

Figure 1: Runtime comparison of Apriori and FP-Tree at different support thresholds

You are provided with a plot in Figure 1 showing the runtime of Apriori and FP-Tree algorithms at minimum support thresholds of 5%, 10%, 25%, 50%, and 90%. You must do the following:

Sub-task 1: Write a script to construct a transactional dataset with approximately 15,000 transactions such that, when `Apriori` and `FP-Tree` are executed on it, their runtime trends qualitatively resemble the given plot. Exact runtime values need not match. The script must be generalized to take the universal itemset and the number of transactions as inputs. Transactions must be exclusively generated by sampling from this item universe. You are not permitted to use shortcuts such as duplicating identical transactions that artificially force the given runtime behavior. [**10 marks**]

Sub-task 2: Re-run *Task 1* on your constructed dataset using the same support thresholds and generate the corresponding runtime plot. [**2 marks**]

Sub-task 3: In your report, compare the results on the constructed dataset with those from the original dataset used in *Task 1*, and explain the observed trends based on the distribution of items across transactions. [**10 marks**]

1.3 Submission guidelines

Your submission folder for this part should look as shown below:

```
q1/
  |-- q1.pdf
  |-- *.py
  |-- q1_1.sh
  |-- q1_2.sh
```

- `q1_1.sh` and `q1_2.sh` should execute the entire pipelines for *task 1* and *task 2*, respectively. See more details in the next section.
- `*.py` indicates your python script(s).
- `q1.pdf` should contain the analysis of your observations comparing the performance characteristics of the algorithms.
- `dataset.dat` is the file containing the generated dataset for *task 2*.

1.4 Auto-grading guidelines

- **Coding environment:**
  - Use the packages given below, for running your code. Using libraries outside this environment will throw an error during evaluation, leading to penalties.

```
- python=3.10
- matplotlib
```

- **Task 1**:
  - Run the following commands:

```
cd A1/q1

bash q1_1.sh <path_apriori_executable> <path_fp_executable> <path_dataset>
<path_out>

# <path_apriori_executable>: absolute filepath to apriori's compiled
executable
# <path_fp_executable>: absolute filepath to fp-tree's compiled executable
# <path_dataset>: absolute filepath to the dataset file
# <path_out>: absolute folderpath where the plot and the outputs at different
thresholds will be saved
```

  - Running this code should run and time the algorithms, generate output files, and generate the plot. After running `q1_1.sh` the output folder should look like the following, where `ap10` corresponds to `Apriori`'s output at 10% threshold:

```
<path_out>
        |-- ap10
        |-- ap25
        |-- ap5
        |-- ap50
        |-- ap90
        |-- fp10
        |-- fp25
        |-- fp5
        |-- fp50
        |-- fp90
        |-- plot.png
```

- **Task 2**:
  - Run the following commands:

```
cd A1/q1

bash q1_2.sh <universal_itemset> <num_transactions>

# <universal_itemset>: superset of distinct items possible across all
transactions
# <num_transactions>: number of transactions in the dataset
```

  - After executing the above code, the newly generated dataset should be available in the current working directory, i.e., `/A1/q1`, with the filename `generated_transactions.dat`.

1.5 Grading

- *Task 1's* output files and the plot will be crosschecked against the TA's outputs and plot.
- TA will re-run *Task 1* for the dataset constructed in *Task 2*, and verify the output files and plot generated.
- Your analysis in the report carries the maximum weightage **[18 marks]**.

1.6 Additional tips

- Use `baadal` for this part.
- You need not submit the binaries or the libraries.
- Don't forget the X and Y labels in your plot and indicate the units of measurement.
- Your output filenames should match the above mentioned file names for the autograder to work.

2 Frequent subgraph mining (20 marks)

2.1 Comparision of FSG, gSpan and FP-Tree [8 (implementation) + 12 (report) marks]

This part will familiarize you with frequent subgraph mining tools. Run `gSpan`, `Fsg` (also known as `PAFI`), and `Gaston` on the `Yeast` dataset at `minSup` = 5%, 10%, 25%, 50%, 95% . You may need to write a script to change the dataset format for each algorithm's library. In a single plot, plot the running times against the minimum supports for each method. In your report, explain the observed trends. Specifically, comment on the growth rates and why one technique is faster. You are advised to consult the respective papers.

- **Resources:** Refer to the README files in the libraries for the runtime arguments and input formats.
  - `gSpan`:
    - ∗ Paper: https://sites.cs.ucsb.edu/~xyan/papers/gSpan.pdf
    - ∗ Binary: https://sites.cs.ucsb.edu/%7Exyan/software/gSpan.htm
  - `Fsg`:

    * Paper: https://ieeexplore.ieee.org/document/1316833
    * Library: OneDrive link. Compile this library using *make* command.
  - Gaston:
    * Paper: https://liacs.leidenuniv.nl/~nijssensgr/gaston/gaston-april.pdf
    * Library: https://liacs.leidenuniv.nl/~nijssensgr/gaston/download.html.
    * Compile the above library using *make* command.
    * In case, you face compilation errors, add `#include <getopt.h>` to `main.cpp` before running `make`.
- **Dataset:**
  - Dataset: OneDrive link
  - Dataset format:

```
#graphID
number of nodes
node label
node label
.
.
.
number of edges
source_node_id, destination_node_id, edge_type
source_node_id, destination_node_id, edge_type
.
.
.
```

2.2 Submission guidelines

Your submission folder for this part should look as shown below:

```
q2/
  |-- q2.pdf
  |-- *.py
  |-- q2.sh
```

- `q2.sh` should execute your entire pipeline. See more details in the next section.
- `*.py` indicate your python script(s).
- `q2.pdf` should contain the analysis of your observations comparing the performance characteristics of the algorithms.

2.3 Auto-grading guidelines

- **Coding environment:**
  - Use the packages given below, for running your code. Using libraries outside this environment will throw an error during evaluation, leading to penalties.

```
- python=3.10
- matplotlib
```

- **Code execution**:
  - Run the following commands:

```
cd A1/q2

bash q2.sh <path_gspan_executable> <path_fsg_executable>
<path_gaston_executable> <path_dataset> <path_out>

# <path_gspan_executable>: absolute filepath to gspan's compiled executable
# <path_fsg_executable>: aboslute filepath to fsg's compiled executable
# <path_gaston_executable>: aboslute filepath to gaston's compiled executable
# <path_dataset>: absolute filepath to the dataset file
# <path_out>: absolute folderpath where the plot and the outputs at different
minimum supports will be saved
```

– Running this code should run and time the algorithms, generate output files, and generate the plot. After running `q2.sh` the output folder should look like this where `gspan10` corresponds to gSpan's output at `minSup = 10%`:

```
<path_out>
        |-- fsg10
        |-- fsg25
        |-- fsg5
        |-- fsg50
        |-- fsg95
        |-- gaston10
        |-- gaston25
        |-- gaston5
        |-- gaston50
        |-- gaston95
        |-- gspan10
        |-- gspan25
        |-- gspan5
        |-- gspan50
        |-- gspan95
        |-- plot.png
```

2.4 Grading

- **[8 marks]** The output files and the plot will be crosschecked against the TA's outputs and plot.
- **[12 marks]** Your analysis in the report carries the maximum weightage.

2.5 Additional tips

- Use `baadal` for this part.
- You need not submit the binaries or the libraries.
- Don't forget the X and Y labels in your plot and indicate the units of measurement.
- Your output filenames should match the above mentioned file names for the autograder to work.

3 Graph indexing (45 marks, competitive)

Graph indexing addresses a fundamental computational bottleneck in managing and querying large-scale graph databases where exhaustive search is prohibitively expensive. In bioinformatics, it supports chemical compound searching and protein structure matching where subgraph queries identify molecular patterns. Social networks leverage graph indexing for real-time friend recommendations and pattern detection, while financial services use it for fraud detection by rapidly identifying suspicious transaction patterns. Similarly, knowledge graphs and recommendation systems depend on efficient graph indexing to navigate complex entity relationships and deliver personalized results.

3.1 Problem Definition

Definition 1 (Labeled graph)**.** Given a set Ω_V, called the set of node labels, and Ω_E called the set of edge labels, a *labeled graph* is graph $G = (V, E)$ along with two label functions $\psi_V : V \to \Omega_V$ and $\psi_E : E \to \Omega_E$, i.e., $G_{(\psi_V, \psi_E)} = (G, \psi_V, \psi_E)$.

In simple terms, this means that a label $\ell_u \in \Omega_V$ is associated with every node $u \in V$, and a label $\ell_e \in \Omega_E$ is associated with every edge $e \in E$. Two nodes (or edge) with the same label are seen as equivalent (mappable to each other) for the purposes of isomorphism.

Problem 2 (Graph Indexing)**.** Given a graph database $D = \{g_1, g_2, \ldots, g_n\}$, where each g_i is a labeled graph, and a query graph q, the task of the *graph indexing* problem is to return the following result set

$$R_q = \{g_i \mid g_i \in D \text{ and } q \subseteq g_i\}$$

where the notation $q \subseteq g_i$ means that q is subgraph isomorphic to g_i.

3.2 Datasets

You will perform efficient graph indexing on two molecular datasets *(database graphs)* available at the following OneDrive link.

The input to your algorithm—*the query graphs*—should be used to generate sets of candidate graphs from these database graphs. For each query graph q, your algorithm should output $C_q := \texttt{alg}(q)$, such that $C_q \supseteq R_q$. The query graphs are available at this OneDrive link. We will also evaluate your implementation against a set of **hidden query graphs** as part of the competitive evaluation.

The dataset format is as follows, with *explanatory comments* for your understanding provided in parentheses.

```
# (new graph)
v 0 1 (node 0, label 1)
v 1 2 (node 1, label 2)
e 0 1 3 (edge from node 0 to node 1, with label 3)
# (new graph)
.
.
.
```

You are advised to read the datasets' research papers for ideas:

- **TUDataset:** TUDataset: A collection of benchmark datasets for learning with graphs
- **Mutagenicity:** Derivation and Validation of Toxicophores for Mutagenicity Prediction
- **NCI-H23:** Comparison of descriptor spaces for chemical compound retrieval and classification

3.3 Approach Overview

Baseline Method: The naïve approach performs subgraph isomorphism testing between the query graph and every graph in the dataset, returning all graphs containing the query as a subgraph. Since subgraph isomorphism is NP-complete, this approach is computationally prohibitive for large databases.

Index-Based Filtering Approach: To improve efficiency, we suggest you to use a feature-based indexing strategy based on *frequent subgraph mining (FSM)*. One strategy is as follows:

1. **Feature Selection and Index Construction:** Mine the top-k (where $k = 50$) most frequent subgraph fragments from the database using FSM. For each graph g_i in the database, construct a binary feature vector $\mathbf{v}_i \in \{0,1\}^k$ indicating the presence or absence of each of the k frequent fragments.
2. **Query Feature Extraction:** Given a query graph q, perform subgraph isomorphism tests against each of the k frequent fragments to generate a binary query feature vector $\mathbf{v}_q \in \{0,1\}^k$.

3. **Candidate Set Generation:** Apply the necessary condition for subgraph isomorphism: if $q \subseteq g_i$, then every fragment present in q must be present in g_i. Formally, a graph g_i is retained in the candidate set C_q if and only if $\mathbf{v}_q \leq \mathbf{v}_i$ (component-wise), i.e., for all fragments f_j where $f_j \subseteq q$, we must have $f_j \subseteq g_i$. Graphs violating this condition are pruned from consideration.

Optimization: You are not restricted to using the top-k subgraphs. Your goal is to design a feature vector that optimizes the metric given below. An effective strategy would generate more discriminative binary feature vectors, that produce a smaller candidate set C_q, for the given query q. You could consider mining more than k subgraphs, and use them to extract k most discriminative graph fragments for building the feature vectors. You should research pre-neural network graph indexing literature for this purpose.

Performance Metric: The quality of the indexing scheme is evaluated by the size of the candidate set $|C_q|$ returned using the selected set of k discriminating subgraph fragments, with smaller candidate sets indicating superior indexing performance. More precisely, the score s assigned to your output is given by

$$s_q = \frac{|R_q|}{|C_q|}.$$

The larger s_q is the better is your algorithm's output for query q.

3.4 Submission format

```
q3/
  |-- convert.sh
  |-- env.sh
  |-- generate_candidates.sh
  |-- identify.sh
  |-- *.py
  |-- q3.pdf
```

Your submission folder for this part should look as above where:

- `identify.sh`
  - It identifies the discriminative subgraphs from the input graphs whose presence/absence will act as features.
  - Only the database graphs will be passed to this script.
  - Ensure that duplicate graphs are removed from the database before passing it to this script. The original ordering of database graphs must be preserved during pre-processing.
- `convert.sh`
  - It converts graphs into feature vectors by performing subgraph isomorphism against the subgraphs stored earlier.
  - This script will be called twice separately, once on the database graphs and once on the query graphs.
  - The feature vectors must be 2D numpy arrays.
- `generate_candidates.sh`
  - It generates the candidate set for each query graph by filtering database graphs based on feature vector compatibility.
  - This script takes as input the query feature vectors (2D numpy array) and database feature vectors (2D numpy array).
  - The output is written to a file named `candidates.dat` where each query graph's serial number is listed after `q #`, followed by the serial numbers of its candidate graphs (space-separated) after `c #`. For example (explanation for your understanding in parenthesis):

```
q # 1
c # 1 2 3 4 5
q # 2
c # 4 5 12 45
q # 3  (Third query graph in the query dataset)
c # 23 46 78 90  (Serial numbers of candidate graphs for query graph 3)
...
...
...
```

– `*.py` indicates your python scripts.
– `env.sh` to setup your code environment.
– `q3.pdf` describes how you identified the discriminative subgraphs and the reasoning behind your approach.

3.5 Running your code

```
bash env.sh
```

```
bash identify.sh <path_graph_dataset> <path_discriminative_subgraphs>

# <path_graph_dataset> : absolute filepath to the dataset of database graphs.
# <path_discriminative_subgraphs>: absolute filepath to store the discriminative
subgraphs
```

```
bash convert.sh <path_graphs> <path_discriminative_subgraphs> <path_features>

# <path_graphs>: absolute filepath to the dataset of graphs. These can be database
graphs or query graphs.
# <path_discriminative_subgraphs>: absolute filepath to the discriminative
subgraphs.
# <path_features>: absolute filepath to store the input dataset mapped to the
feature space. This must be a 2D numpy array.
```

```
bash generate_candidates.sh <path_database_graph_features>
<path_query_graph_features> <path_out_file>

# <path_database_graphs>: absolute filepath to the 2D numpy array of database
graphs dataset.
# <path_query_graphs>: absolute filepath to the 2D numpy array of query graphs
dataset.
# <path_out_file>: absolute filepath to the file for storing candidate sets for
the query graphs.
```

Algorithmic Restrictions

- You are allowed to use libraries and are not limited to Python libraries.
- Your features should not be any dense representations of graphs. They should be binary indicating the presence/absence of discriminative subgraphs.
- Do not use neural network based approaches for this part. Instead, research the graph indexing literature from the pre-graph-neural-network era.

- The goal of this assignment is to identify the discriminative subgraphs. Your algorithm should be tailored for this.
- If you are in doubt about the validity of your algorithm choice, ask in private on Piazza for confirmation.
- Your code will be crosschecked against your report to catch any cheating attempts.

Code Environment

- You have the freedom to build your environment in this part.
- Hence, you'll have to submit a script `env.sh` that creates the necessary environment.
- You'll have internet access while setting up the environment.

Grading

- Report **[10 marks]**
- Competitive:
  - `Part 1:` Visible query graphs **[15 marks]**
  - `Part 2:` Hidden query graphs **[20 marks]**

‼️ Competitive Scoring Mechanism

Let Q be the set of query graphs. We want to discourage large candidate sets. Towards this end, we propose to use the following marking scheme. For each student $\in$ class, for each query $q \in Q$, do the following:

1. **Identify the best score:** Identify the best score achieved by any student on this query as

$$s_{\text{best}} = \max_{\text{student} \in \text{class}} s_q^{\text{student}}$$

2. **Identify a low reference score:** We choose the score at the 5-percentile from top, denoted $s_{5\%}$.
3. **Assign marks out of 100:** We then assign marks out of 100 as follows.

$$\text{marks}_{100}(s_q^{\text{student}}) = 50 + 50 \cdot \frac{\ln s_q^{\text{student}} - \ln s_{5\%}}{\ln s_{\text{best}} - \ln s_{5\%}}$$

   This ensures that more than 95% of students will achieve marks above 50 in any particular query. We use the natural logarithm (ln) to make the effect of the size of $|C_q|$ linear (because otherwise due to the compression-effect of the $1/x$ function, two large candidate sets will achieve similar score).

Final marks for a student will be assigned by averaging over queries and rescaling:

$$\textbf{marks}(\text{student}) = \text{max-marks} \cdot \frac{1}{|Q|} \sum_{q \in Q} \text{marks}_{100}(s_q^{\text{student}})$$

If the candidate sets are too large for all queries, the above formula may result in negative marks. In that case, the marks will be clamped to 0. The value of s_{best} used in the above formula will be fixed during the first pass of grading, in case a submission's code works in re-grading and surpasses the best performance there marks will still be clamped at the maximum without affecting other scores.

Tips

- Needless to say that you'll have read-only access to the data while grading. Any attempt to tamper with it will throw an error and show up in the logs. Strict action will be taken in such cases.