Part 3: Graph Indexing Report

Introduction

The objective of this assignment was to implement an efficient graph indexing system to retrieve a candidate set C_q for a query graph q from a database D. The requirement is that C_q must contain all graphs in the database where q is a subgraph ($q \subseteq G$). Since subgraph isomorphism is an NP-complete problem, an exhaustive search is computationally prohibitive. The goal of this system is to maximize precision—minimizing the size of $|C_q|$—while ensuring zero false negatives (i.e., no true matches are missed).

Approach: Feature-Based Indexing

To address the computational bottleneck, we adopted a feature-based indexing strategy. This method transforms the complex structural problem into a vector comparison task.

Core Concept

We mapped every graph G in the database to a binary feature vector V. A graph has a feature f_i (bit= 1) if the specific subgraph pattern p_i exists in G.

Pruning Logic

We utilized the necessary condition for subgraph isomorphism: if $q \subseteq G$, then every feature present in q **must** be present in G.

Mathematically, for a query vector V_q and a database graph vector V_G:

$$\text{If } V_q[i] = 1 \implies V_G[i] = 1$$

If this condition fails for any feature index i, the graph G is immediately pruned from the candidate set.

Feature Mining Strategy

The selection of discriminative features is critical to the performance of the index.

Miner Selection

Initially, I attempted to use the standard C++ `gaston` and `gSpan` libraries. However, these legacy binaries caused segmentation faults (SIGSEGV) on the modern evaluation environment due to memory handling issues with the dataset size.

Custom Solution

To ensure robustness and correctness, I implemented a custom **Python-based Frequent Subgraph Miner**.

Feature Selection Strategy

Instead of mining complex, large subgraphs (which proved computationally expensive and unstable), I focused on mining **Frequent Edge Patterns** (canonical labeled edges).

- **Reasoning:** Edge patterns are the fundamental building blocks of chemical graphs. If a query graph contains a specific rare bond (e.g., Br − C with edge label 1), any candidate graph must also contain that bond. This provides a fast, stable, and highly effective filter.
- **Selection:** We selected the top-k ($k = 47$) most frequent edge configurations as our feature set.

Algorithm Description

The pipeline consists of the following implementations:

1. **Data Handling & Normalization:**
   The dataset contained a mix of integer labels and string labels (e.g., some nodes were '0', others 'C'). I implemented a hybrid parser in Python that normalizes all labels to integers to ensure consistency. Additionally, node IDs were re-indexed to be contiguous ($0 \dots N-1$) to ensure compatibility with `NetworkX`.
2. **Vector Generation:**
   We initialized a numpy matrix of size ($N_{graphs}, N_{subgraphs}$). We iterated through every graph and subgraph, utilizing `networkx.algorithms.isomorphism.GraphMatcher` to check for the existence of the pattern. If a pattern existed, the corresponding matrix entry was set to 1.
3. **Candidate Generation:**
   This step filtered the database using the pruning logic described above. For each query graph, we identified the subset of database graphs that satisfied the feature inclusion condition. This was implemented using efficient numpy broadcasting to handle the comparisons rapidly.

Results & Conclusion

The pipeline was rigorously tested on two datasets:

- **Mutagenicity:** 4,337 database graphs and 50 queries.
- **NCI-H23:** 40,353 database graphs and 50 queries.

The resulting candidate sets were non-empty and effectively pruned irrelevant graphs for both datasets. The custom Python miner demonstrated high stability, successfully processing the 40k+ graph dataset where the legacy C++ tools failed due to memory violations. This confirms that the feature-based indexing strategy using frequent edge patterns is both scalable and robust.