Q3: Graph Indexing

Discriminative Subgraph Selection for Efficient Query Processing

1 Overview

This report describes the implementation of an efficient graph indexing system for subgraph query processing. The system employs a feature-based indexing strategy that uses frequent subgraph mining to identify discriminative graph patterns, which are then used to quickly filter candidate graphs for a given query.

The approach uses **Gaston** for frequent subgraph mining at a 70% support threshold, selecting the top-50 most frequent patterns as discriminative features. These patterns are then used to construct binary feature vectors for both database and query graphs, enabling efficient candidate set generation through component-wise filtering.

2 Approach

Our graph indexing system consists of three main components: discriminative subgraph identification, feature vector construction, and candidate set generation. Each component is designed to maximize query processing efficiency while maintaining correctness.

2.1 Discriminative Subgraph Identification (identify.py)

The first step in building our index is to identify discriminative subgraph patterns from the database. The process consists of the following steps:

1. **Data Preprocessing:** Load the database graphs and remove duplicate graphs while preserving the original ordering. The `remove_duplicate` function creates a set of graph tuples to track seen graphs, ensuring each unique structure appears only once. This reduces redundant computation in subsequent steps.

2. **Format Conversion:** Convert the cleaned database to Gaston's input format using the `gaston_format` function. Gaston expects graphs where each graph starts with `# 1` and `t graph_id`, followed by vertex definitions (`v node_id label`) and edge definitions (`e src dst label`).

3. **Frequent Subgraph Mining:** Run Gaston with a support threshold of 70%. The support frequency is calculated as

$$\max\left(1, \left\lfloor \frac{70}{100} \times \text{total\_graphs} \right\rfloor\right).$$

This high threshold ensures we mine patterns that appear in the majority of graphs, providing features with good coverage across the database.

4. **Top-k Selection:** Extract the top 50 subgraphs by support count using `extract_topk`. Patterns are parsed from Gaston's output, sorted by support in descending order, and the top 50 are written to the output file. Each pattern includes its support count as a comment for reference.

Rationale: Using a high support threshold (70%) focuses on patterns that are frequent enough to be useful for filtering but not so universal that they appear in every graph. These patterns provide the best balance between coverage and discriminative power. The top-50 selection ensures sufficient features to distinguish between graphs while keeping the feature space manageable for efficient subgraph isomorphism checking.

2.2 Feature Vector Construction (convert.py)

Once discriminative subgraphs are identified, we construct binary feature vectors for both database and query graphs as follows:

1. **Graph Loading:** Parse both the input graphs (database or query) and the discriminative subgraphs using the `parse_graph` utility from `utils.py`. This parser handles the standard graph format with `#` delimiters and `v`/`e` prefixes.

2. **Graph Conversion:** Convert all graphs to NetworkX format using `make_networkx_graph`. Nodes are added with labels as attributes, and edges are added with labels as attributes.

3. **Subgraph Isomorphism Testing:** For each graph G and each discriminative subgraph S, check whether S is subgraph isomorphic to G using NetworkX's `GraphMatcher` with custom node and edge label matching. The `subgraph_is_isomorphic()` method returns true if the pattern exists in the target graph.

4. **Feature Matrix Construction:** Construct a binary feature matrix F using NumPy, where $F[i][j] = 1$ if subgraph j is present in graph i, and 0 otherwise. The matrix is saved as a `.npy` file for efficient storage and retrieval.

Implementation Details: Exact label matching is enforced for both nodes and edges to preserve semantic correctness, ensuring that structurally similar but semantically different patterns are treated as distinct features.

2.3 Candidate Set Generation (generate_candidates.py)

The final step uses feature vectors to generate candidate sets for query graphs, significantly reducing the search space before expensive exact subgraph isomorphism testing.

Filtering Rule: A database graph g is retained as a candidate for query q if and only if every feature present in q is also present in g. This is implemented as:

$$\texttt{np.all(query\_feature} \leq \texttt{db\_feature).}$$

Correctness Guarantee: This rule is based on a necessary condition for subgraph isomorphism: if $q \subseteq g$, then every structural pattern present in q must also appear in g. Hence, the candidate set contains all true matches, ensuring no false negatives.

Output Format: For each query at index i, the output contains `q # i` followed by `c #` and a space-separated list of candidate graph indices (0-indexed), as required by the assignment specification.

3 Algorithm Analysis

3.1 Time Complexity

- **Identification Phase:** $O(\text{Gaston\_runtime} + n \times k)$, where n is the number of database graphs and $k = 50$ is the number of features.
- **Conversion Phase:** $O(n \times k \times T_{\text{iso}})$, where T_{iso} is the cost of subgraph isomorphism between a small pattern and a database graph.
- **Query Phase:** $O(|Q| \times n \times k)$, which is significantly faster than naive subgraph isomorphism with cost $O(|Q| \times n \times T_{\text{exact}})$.

3.2 Space Complexity

The space complexity is $O(n \times k)$ for storing the feature matrix, plus $O(k \times |S|)$ for storing k discriminative subgraphs of average size $|S|$. Since $k = 50$ and subgraphs are small, the overhead is minimal.

4 Design Decisions and Rationale

4.1 Why 70% Support Threshold?

Very frequent patterns ($> 90\%$) appear in nearly all graphs and provide little discriminative power, while very rare patterns ($< 10\%$) appear in too few graphs to be broadly useful. A 70% threshold balances coverage and discriminative ability.

4.2 Why Top-50 Patterns?

Choosing $k = 50$ balances expressiveness and computational cost. Too few features result in weak filtering, while too many increase indexing cost. Empirically, 50 features provide effective discrimination with manageable overhead.

4.3 Why Gaston?

Gaston uses an efficient depth-first search strategy for mining connected subgraphs, making it well suited for identifying structural motifs useful for graph indexing. It provides a good balance of speed and pattern quality compared to alternatives such as gSpan or FSG.

4.4 Component-wise Filtering Strategy

The filtering rule is sound and complete: it guarantees no false negatives while allowing some false positives, which would be resolved by subsequent exact verification in a complete system.

5 Conclusion

This implementation presents an effective graph indexing system based on discriminative subgraph features. By leveraging frequent subgraph mining with Gaston and careful feature selection, the system significantly reduces candidate set sizes while guaranteeing correctness. The approach balances efficiency and effectiveness and provides a strong baseline for competitive graph indexing tasks.