COL761 A1 — Question 3 Report
Date: 2026-02-03

This report describes the exact approach implemented in the submitted code for Q3 (graph indexing / candidate generation). The goal is to generate a *safe* candidate set C_q for every query q such that it contains all true matches (0 false negatives), while keeping C_q as small as possible for a higher score.

Files submitted

- **identify.sh**, **identify.py**: builds a feature-definition file from the database graphs.
- **convert.sh**, **convert.py**, **graph_utils.py**: converts graphs into a binary feature matrix saved as .npy.
- **generate_candidates.sh**, **match.py**: produces candidates.dat using feature containment check.
- **env.sh**: creates a venv and installs dependencies.

Dependencies / libraries

- Python 3
- numpy (vectorized feature matrix operations)
- networkx (parsing and basic graph utilities such as cycle detection)
- No GPU / ML frameworks are used; features are sparse and binary as required.

Input / output interfaces (matches the pdf)

- **identify.sh <path_graph_dataset> <path_discriminative_subgraphs>** → writes a feature-definition file at the given output path.
- **convert.sh <path_graph_dataset> <path_discriminative_subgraphs> <path_features>** → writes a NumPy array (N×F) to path_features.
- **generate_candidates.sh <path_database_features> <path_query_features> <path_candidates>** → writes candidates.dat to path_candidates.
- All scripts accept absolute/relative paths and do not assume any fixed filenames.

High-level idea

If q is a subgraph of g, then many *monotone* properties of q must also hold for g. We encode each graph as a binary vector v(G) of such monotone properties (features). During candidate generation, we keep g as a candidate for query q iff:

v(g) ≥ **v(q) componentwise** (i.e., every 1-bit in v(q) is also 1 in v(g)).

This guarantees **0 false negatives** as long as every feature is monotone under subgraph embedding. It may keep extra graphs (false positives), which is acceptable — score improves as candidates shrink.

Feature design

All features are **binary**. Conceptually, each feature corresponds to the presence of a small fragment or a safe lower-bound constraint. The implemented feature set is the union of the following groups:

1) Size lower-bounds

- NV>=k: graph has at least k vertices (k in a small set of thresholds).
- NE>=k: graph has at least k edges.

Monotonicity: if $q \subseteq g$, then $|V(q)| \le |V(g)|$ and $|E(q)| \le |E(g)|$, so any satisfied threshold in q is also satisfied in g.

2) Node-label multiplicity lower-bounds

- A:L>=k: number of nodes with label L is at least k.

Monotonicity: injective node mapping preserves labels, so g must contain at least as many nodes of each label as q.

3) Edge-type multiplicity lower-bounds

- E:(uLabel-edgeLabel-vLabel)>=k: number of edges of a given typed form is at least k (endpoint labels are sorted for undirected edges).

Monotonicity: each edge in q maps to a distinct edge in g with the same labels, so counts cannot decrease.

4) Labeled degree lower-bounds

- D:L>=k: there exists a node with label L whose (graph) degree is at least k.

Monotonicity: in a subgraph embedding, a node's degree in g is at least its degree inside the embedded subgraph, so if q has a label-L node with degree ≥k, the image in g also has degree ≥k.

5) Cycle presence

- CY:any: graph contains at least one cycle (detected via NetworkX cycle_basis).

Monotonicity: if q contains a cycle, that cycle's edges must appear in g, so g also contains a cycle.

6) Hashed structural fragments with count thresholds (strong pruning)

To get stronger pruning than global counts alone, we also include features derived from small local fragments. These are still **safe** because they represent lower bounds on the number of fragment occurrences.

H2: edge-aware wedges (length-2, center node with two incident edges).

H3: edge-labeled simple paths of length 3 (4 nodes, 3 edges).

HS: edge-aware stars (neighbor-type multiplicities around a center label).

Because the number of distinct fragment signatures can be large, we map each signature (plus its threshold) into a fixed bucket via a stable MD5 hash. This introduces collisions, which can only create *more* candidates (false positives), never false negatives.

Why the hashed fragment features are monotone

Each fragment instance in q maps to a corresponding fragment instance in g under an injective node mapping. Therefore, for any fixed signature S, count_g(S) ≥ count_q(S). We then set a binary feature 'S>=t' to 1 iff count(S) ≥ t. Since count_g(S) ≥ count_q(S), if q satisfies 'S>=t' then g also satisfies it. Hashing does not change this implication: if q triggers a bucket b (because some S hashed to b), then g also triggers the same bucket because it contains the same S.

Candidate generation rule

Given database feature matrix M_{db} (N×F) and query feature matrix M_q (Q×F), for each query q we select candidates:

$C_q = \{ i \mid M_{db}[i, j] \geq M_q[q, j] \text{ for all features } j \}$

This is implemented in **match.py** using a vectorized NumPy check (np.all(db_matrix >= q_vec, axis=1)). The output is written in the exact required format:

```
q # <query_serial>
c # <space-separated 1-based db serials>
```

Correctness / 0 false negatives guarantee

Let R_q be the true result set for query q (graphs in DB that contain q as a subgraph). We must ensure $R_q \subseteq C_q$.

For any $g \in R_q$, q is a subgraph of g. By construction, every individual feature is monotone, so $v(q) \leq v(g)$ componentwise. Therefore g passes the containment test and is included in C_q. Hence, there are **zero false negatives**.

Why this helps pruning (smaller candidates.dat)

The stronger the feature set, the more likely a random database graph violates at least one query feature bit, and gets filtered out. In practice, the hashed structural fragments (H2/H3/HS) add many independent constraints beyond simple label counts, which substantially reduces candidate sizes while staying safe.

Notes on ordering and duplicates

- Candidates are output in increasing order of database index (1-based), which matches 'original ordering' requirement.
- identify.py also deduplicates the database internally while preserving first occurrence order. This does not affect correctness; it only affects which label universe is used to enumerate features.

How to run

- bash env.sh
- bash identify.sh data/muta_graphs.txt features.txt
- bash convert.sh data/muta_graphs.txt features.txt db_vectors.npy
- bash convert.sh data/query_graphs.txt features.txt q_vectors.npy
- bash generate_candidates.sh db_vectors.npy q_vectors.npy candidates.dat