COL761 A1 — Question 3 (Graph Indexing) Report

Date: 2026-02-03

1 What we need to do (and what can go wrong)

For every query graph q, we must output a candidate set C_q of database graph IDs. The **only hard correctness requirement** is:

$$R_q \subseteq C_q \quad \text{where } R_q = \{g_i \in D \mid q \subseteq g_i\}.$$

So: **zero false negatives**. Even one missed true match breaks the score for that query.

At the same time, the competitive score prefers smaller candidate sets, so the whole game is: prune as much as possible, but *only using conditions that are guaranteed necessary for subgraph isomorphism.*

2 Files submitted and the exact I/O contract

The folder follows the required submission structure (scripts + python + report). In particular, the pipeline is:

```
bash env.sh
bash identify.sh <path_db_graphs> <path_discriminative_subgraphs>
bash convert.sh <path_graphs> <path_discriminative_subgraphs> <path_features.npy>
bash generate_candidates.sh <db_features.npy> <query_features.npy> <candidates.dat>
```

The output file is exactly `candidates.dat` with the required format:

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

3 High-level idea (safe pruning rule)

I represent every graph G as a binary feature vector $v(G) \in \{0,1\}^F$.

The key constraint I enforce is:

$$q \subseteq g \;\Rightarrow\; v(q) \leq v(g) \quad \text{(component-wise)}.$$

Then candidate generation is very simple:

$$C_q \;=\; \{\, i \mid v(g_i) \geq v(q) \text{ component-wise} \,\}.$$

This is implemented in `match.py` using vectorized NumPy: `np.all(db_matrix >= q_vec, axis=1)`.

Important: This check can produce false positives (extra candidates), which is acceptable. It must *never* produce false negatives, which is what monotonicity guarantees.

4 Feature design (what bits mean and why they are safe)

All features are binary. Conceptually, each bit corresponds to a *necessary fragment/property*: if a query has it, any supergraph must also have it.

I use a union of several feature groups. The first few are "global" (fast and cheap), and the last group (hashed local fragments) is what usually gives the strongest pruning.

(1) Size lower bounds

- `NV>=k`: at least k vertices.
- `NE>=k`: at least k edges.

Why safe: any embedding $q \subseteq g$ is injective, so $|V(q)| \leq |V(g)|$ and $|E(q)| \leq |E(g)|$.

(2) Node-label multiplicity lower bounds

- `A:L>=k`: at least k nodes with label L.

Why safe: the mapping preserves node 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`: at least k edges of a typed form (endpoint labels sorted for undirected edges).

Why safe: edges in q map to distinct edges in g with the same labels, so typed-edge counts cannot decrease.

(4) Labeled degree lower bounds

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

Why safe: in the supergraph, the image of a node has degree at least the degree it had inside the embedded subgraph.

(5) Cycle presence

- `CY:any`: the graph contains at least one cycle.

Why safe: if q contains a cycle, all those cycle edges must appear in g, so g must be cyclic too.

(6) Hashed local structural fragments with count thresholds (main pruning power)

To get much stronger pruning than label/edge counts alone, I also include features from small local fragments, but still as *lower bounds on occurrence counts.*

- **H2**: edge-aware wedges (length-2 patterns centered at a node).
- **H3**: edge-labeled simple paths of length 3 (4 nodes, 3 edges).

- **HS**: edge-aware stars that summarize neighbor-type multiplicities around a center label.

Each fragment instance in q must map to a corresponding fragment instance in g, therefore for any fixed signature S: $\text{count}_g(S) \geq \text{count}_q(S)$. I then turn this into binary features of the form "$S \geq t$".

Why hashing is still safe. The number of distinct signatures can be large, so I hash signatures into fixed buckets using a stable MD5 hash. Collisions can only merge different signatures into the same bucket. That can only cause *extra* candidates (false positives), never missing true matches: if q has some signature S then g also has S, so both hit the same bucket.

5 Correctness proof (0 false negatives)

Take any true match $g \in R_q$. Then $q \subseteq g$. Every individual feature bit I use is monotone under subgraph embedding, so $v(q) \leq v(g)$. Therefore g passes the containment check and is included in C_q. Hence, the candidate generator has **zero false negatives**.

6 Why this reduces `candidates.dat` size (and improves score)

The marking scheme rewards larger $|R_q|/|C_q|$, i.e., smaller candidate sets. Every additional independent monotone constraint increases the chance a random database graph violates at least one required bit from q. In practice, the hashed fragment features (H2/H3/HS) add structural constraints beyond simple label counts, and that is what tends to shrink the candidate sets noticeably while remaining fully safe.

7 Duplicates and ordering

- The assignment requires removing duplicate DB graphs while preserving original ordering during preprocessing. `identify.py` performs this by keeping the first occurrence of each signature.
- Candidate IDs are written in increasing 1-based order, which matches the "original ordering" requirement.

8 How to run (local)

```
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
```