COL761/COL7361/AIL7026 Data Mining

Assignment 1: Frequent Itemset Mining

Team T123

Kislaya Shrestha	2025SIY7573
Palaash Goel	2025AIZ8825
Anoop Parashar	2025MCS2118

February 3, 2026

Task 3: Graph Indexing

This task required us to develop a strategy to select discriminative features from a database of graphs so that encoding them as binary feature vectors will lead to minimally-sized candidate sets for different query graphs.

Methodology

Step 1: Frequent Subgraph Mining

In order to identify discriminative features from a given graph database, we make a simple but important observation. If a subgraph is too discriminative while also being extremely rare, it will be an excellent feature for queries containing that subgraph. However, due to its rarity, it would be infeasible to rely on its appearance in a query, thereby effectively leading to a rather under-utilized feature. On the other hand, if a subgraph is too common, its discriminative power is reduced since there is a large probability that it will be present in any given query and database graph, leading to minimal filtering of the database. Therefore, we hypothesize that we must balance discriminative power and frequency in order to extract optimal features that will ultimately lead to small candidate sets.

In order to achieve this, we employ gSpan, a depth-first frequent subgraph mining algorithm that utilizes a pattern-growth strategy to mine frequent subgraphs. Specifically, we mine frequent subgraphs from the databse at a support threshold of 20%. While this may seem low, the objection is to maximize the number of subgraphs we mine from a database while also ensuring that we filter out highly rare subgraphs. Once mined, we order the frequent subgraphs in decreasing order of available support and retain only the top 100 of them. This results in a fairly-sized pool of candidates which we can prune further to obtain our final set of discriminative features.

Step 2: Discriminative Feature Filtering

In order to filter the set of frequent subgraphs, we compute the discriminative ratio (γ) for each subgraph, as proposed by Yan et al. (2004). Formally, we compute γ as:

$$\gamma = \frac{|\bigcap_i D_{f_{\phi_i}}|}{|D_x|} \tag{1}$$

where x is a frequent subgraph mined in the first stage (but not yet a feature) and $D_{f_{\phi_i}}$ is the set of database (D) graphs to which feature f_{ϕ_i} is subgraphic isomorphic ($\{g : f_{\phi_i} \subseteq g \wedge g \in D\}$), given that it is also subgraph isomorphic to x ($f_{\phi_i} \subseteq x$). Additionally, D_x is the set of all database graphs to which x is subgraph isomorphic. As per Yan et al. (2004), $\gamma \geq 1$ wherein a high γ corresponds to a graph of high discriminative power and thus, high fidelity for acting as a feature. On the other hand, a low γ value represents a fragment that can completely (or approximately) be indexed by already selected features and thus, need not be added to the feature set. Following this, we iterate through all 100 previously identified frequent subgraphs and at each step of iteration, choose the graph with the highest γ value. We repeat this process until we have selected 50 discriminative features.

Conversion to Binary Features and Selection of Candidate Sets

Once selected, we convert each database graph g into binary vectors v_g where the i^{th} component v_{g_i} is set to 1 if $f_i \subseteq g$ and 0 otherwise. Now, for a query graph q, we first convert it to its binary vector (v_q). The candidate set for q contains all database graphs g for which $v_q \leq v_g$.

Note that, in order to minimize the cases where $v_q \leq v_g$, we sort all selected features in increasing order of support. This has a highly probably effect of shifting the first '1' in v_g rightward (features with higher supports are highly common), leading to a higher probability of the aforementioned candidate selection criterion to be violated and the resulting candidate set to be even smaller.

Implementation Details

We utilize the `rustworkx` (Treinish et al., 2022) and `igraph` (Antonov et al., 2023) Python libraries to operate efficiently on large graph databases. Additionally, we utilize the `multiprocessing` library wherever applicable to leverage parallel processing for higher efficiency.

References

M. Antonov, G. Csárdi, S. Horvát, K. Müller, T. Nepusz, D. Noom, M. Salmon, V. Traag, B. F. Welles, and F. Zanini. igraph enables fast and robust network analysis across programming languages, 2023. URL `https://arxiv.org/abs/2311.10260`.

M. Treinish, I. Carvalho, G. Tsilimigkounakis, and N. Sá. rustworkx: A high-performance graph library for python. *Journal of Open Source Software*, 7(79):3968, Nov. 2022. ISSN 2475-9066. doi: 10.21105/joss.03968. URL `http://dx.doi.org/10.21105/joss.03968`.

X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 335–346, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138598. doi: 10.1145/1007568.1007607. URL `https://doi.org/10.1145/1007568.1007607`.