Enhanced Subgraph Matching by Spectral Pruning

Author: Zhiwei Zeng
guilinabaoge@gmail.com
Aarhus University

Supervisors: Panagiotis Karras, Davide Mottin

ABSTRACT

The state-of-the-art approach for solving subgraph matching problems is preprocessing enumeration-based. However, the bottleneck for this approach is the pruning power of the offline local filter in the filtering method. To improve the pruning power of the local filter, we utilize the local structure around the vertex. We encode the local structure of a vertex, denoted as 'v', into a numerical space by computing the top 10 eigenvalues of v's 2-hop spanning tree's Laplacian matrix. These top 10 eigenvalues are known as the signature for 'v'. The spectral filter utilizes the vertex signatures to filter false positive candidates. Based on our experiments, we have found that the spectral filter performs best in scenarios where label information is limited for pruning. However, in scenarios where label information is rich for pruning, the spectral filter only provides a very limited increase in improvement, which does not justify the overhead spent on the extra filtering stage. In conclusion, we recommend using spectral filters in scenarios where label information is lacking, for example, in a data graph where most labels are the same. The source code for this thesis is available at : https://github.com/Guilinabaoge/SubgraphMatching_Spectral

1 INTRODUCTION

Subgraph matching is a fundamental problem in graph theory and computer science. It involves finding occurrences of a smaller graph (the query graph) within a larger graph (the data graph). Given a query graph q and a data graph G subgraph matching is to find all subgraphs in G that are isomorphic to q. Subgraph matching has various applications across different domains. Here are a few examples :

Social Network Analysis : Subgraph matching is used to identify communities or groups of individuals with similar interests or relationships in social networks. By finding common subgraphs, analysts can understand patterns of social interactions, identify influential nodes, or detect anomalies.

Chemical Compound Analysis : In chemistry, subgraph matching is used to analyze molecular structures. By representing molecules as graphs, researchers can identify substructures or subgraphs that are responsible for specific chemical properties or reactions. This information aids in drug design, compound similarity analysis, and chemical synthesis planning.

Network Security : Subgraph matching can be utilized in network security to detect patterns or signatures of malicious activities within network traffic. By representing network flows as graphs, suspicious subgraphs can be matched against a database of known attack patterns, helping in intrusion detection and prevention.

The solution to the subgraph matching problem empowers us to delve into graph data, conduct comprehensive analyses, and derive valuable insights across diverse fields, thereby driving advancements. However, subgraph matching represents a formidable challenge in terms of achieving efficient solutions due to its inherent complexities. The subgraph matching problem is NP-complete because it includes a subgraph isomorphism test, which is an NP-complete problem.[5]

The number of operations needed to check the isomorphism of two n-node graphs using a brute-force approach is $n!$. For two 32-node graphs, in the worst-case scenario, it would require approximately $2.6310^{3}5$ operations to check isomorphism. Performing such a large number of operations within a reasonable time frame would be practically impossible.

Therefore subgraph matching algorithms are employed heuristics to check for isomorphism. These algorithms aim to reduce the search space by pruning false positive embedding candidates (vertex in the data graph that can't be mapped by any vertex in the query) using specific properties(spacial/label property) of the vertex to speed up the process. While they cannot guarantee a constant or polynomial time complexity, they are designed to handle larger graphs more efficiently than brute-force methods.

A common way to answer a subgraph query is to use the Filter and Verification framework(FV). The main idea of this approach is to prune as many false positive vertices from the candidate space, to reduce the number of subgraph isomorphic tests, which is the most expensive procedure. Recently, the Filter and Verification approach is not considered an effective approach. The reason is that its pruning power has been overestimated and it has a large memory and running time needs. It is overestimated in the sense that all the algorithms were paired with simple enumeration algorithms like Ullmanns [19] or VF2[14], where the initial pruning helped to improve the running time. When the FV algorithms were paired with more sophisticated enumeration algorithms, the benefit from filtering was limited[13]. This led to the conclusion that index-based pruning removes only simple instances that can be easily removed by good backtracking algorithms and the subgraph matching community gradually dropped the FV framework.

The state-of-the-art graph-matching algorithms now use a preprocessing enumeration framework[1, 3, 10]. Similar to the FV framework the idea is to prune the search space before the subgraph isomorphic test. The framework will create an auxiliary data structure for the candidate vertex space of the query graph. The difference is FV algorithms do enumeration on the original data graph, while preprocessing-enumeration algorithms do enumeration on the auxiliary data structure of the candidate vertex set of the query graph.

The candidate vertex set for each vertex in the query graph will be pruned based on a pruning strategy during the construction of the auxiliary data structure. Each algorithm has developed its

own pruning strategy based on different graph properties. The Filtering process usually has two stages ; local pruning and global refinement.

The bottleneck of the preprocessing-enumeration-based graph-matching algorithm is the local pruning. Before building the auxiliary data structure each algorithm applies a static local pruning to reduce the initial candidate space size. The weak pruning power of local pruning will lead to a big auxiliary data structure that is close to the original data graph, which makes the process of building such an auxiliary structure meaningless and waste the overhead to build $\mathcal{A}$. Therefore a local pruning strategy with strong pruning power is significant for the performance of preprocessing-enumeration-based algorithms. The existing local pruning strategies are usually very label-dependent. This becomes a problem when the label information in the data graph is limited, the local filter will lose most of its pruning power. This prohibits the online refinement stage. When the initial pruning performs poorly the auxiliary structure will create a large candidate vertex space for each query vertex and makes state-of-the-art algorithms unsuitable. (pruning 1 offline can lead to pruning 10 online). In this work, we proposed to use additional structure information in the local pruning process, the spectral filtering strategy.

The main idea behind the spectral filtering strategy is when the query graph q is subgraph isomorphism to a data graph G, for each vertex u in q, there must exist a corresponding vertex v in G. Furthermore, the local structure around u in q should be preserved around v in G. We can prune v from $c(u)$ if the local structure of v and u are different. However, directly checking the local structure is a very expensive operation, since it needs to perform a structured comparison. Previous work[**?**] proposed the method GCoding which encodes the structures of a graph into a numerical space, by using Gcoding we can reduce the checking cost significantly, due to numerical comparison instead of structure comparison.

The Spectral Filtering Strategy builds an index of vertex topology signature for each vertex in the query and data graph. With the index, we can perform pruning without using the data graph. We can prune v from $C(u)$ if the vertex topology signature of vertices v and u are not compatible.

In summary, we made the following contributions in this thesis :

(1) We improved the local pruning of preprocessing-enumeration algorithms using a Spectral Pruning Strategy.

(2) We compare and analyze the performance of 6 different preprocessing-enumeration-based algorithms after enhancing their local pruning with spectral pruning strategy.

(3) We conduct experiments for each algorithm with real-world datasets with queries containing different levels of label information by introducing wildcard labels in the queries.

2 BACKGROUND

In this section, we will give a formal definition of related concepts. Table 1 summarised the notations used in this thesis.

2.1 Preliminaries

In this thesis, we focus on the undirected labeled graph $g = (V, E)$. Where V is a set of vertices and E is a set of edges.

Notations	Descriptions
$L(u)$	Label of vertex u
$d(u)$	Degree of vertex u
$N(u, l)$	Neighbour vertices of vertex u has label l
$C(v)$	Candidate vertex set of $v \in q$
q and G	query graph and data graph
ϕ	The matching order
M	Store mappings of $u \in q$
$N_+^\delta(u)((N_-^\delta(u)))$	Backward(forward) neighbors of u given ϕ
$\mathcal{A}$	The auxiliary data structure
$\mathcal{A}^u_{u'}(v)$	neighbors of v in $C(u'))$ where $v \in C(u))$
$N(u)$	Neighbor of vertex u
LC	The local candidates set
C	The candidate set

Table 1 : Notations

Definition 1. *Induced subgraph isomorphism : preserves non-adjacency, if two vertexes are not neighbors in q then they must not be neighbors in the match. A match is a subgraph means a subgraph in the data graph that is isomorphic.*

Definition 2. *Subgraph Isomorphism : Given $q = (V, E)$ and $G = (V'E')$ a subgraph isomorphism is an **injective function** f from V to V' such that (1) $\forall u \in V, L(u) = L(f(u))$; and (2) $\forall e(u, u') \in e(f(u), f(u')) \in E$ [17]*

Problem 3. *Given query graph q and data graph G, subgraph matching is to find all subgraph isomorphisms from q to G. For brevity, we call a subgraph isomorphism a match. We assume that q is connected and $|V(q) \leq 3|$ because finding all matches of a single vertex or a single degree is trivial.*

Algorithm 1 Generic Subgraph Matching Algorithm [17]

```
1:  Input : a query graph q and data graph G
2:  Output : all matches from q to G
3:  C, A ← generate candidate vertex sets and build auxiliary data
    structure
4:  φ ← generate a matching order
5:  Enumerate(q, G, C, A, , 1)
6:  procedure Enumerate(q, G, C, A, M, i)
7:      if i = |φ| + 1 then return M
8:      u ← select an extendable vertex given φ and M
9:      LC(u, M) ← Compute local candidates
10:     for each v ∈ LC(u, M) do
11:         if v ∉ M then
12:             Add(u, v) to M
13:             Enumerate(q, G, C, A, M, i + 1)
14:             Remove(u, v) from M
```

Sun et al.[17] abstract and define preprocessing-enumeration subgraph matching algorithms into three components : Filtering method, Ordering Method and Enumeration Method. In the Filtering method, each $u \in V(q)$ will generate a candidate vertex set $C(u)$. Algorithm 1 presents the generic preprocessing-enumeration algorithm, which takes q and G as input and outputs all matches

from q to G. For each vertex, $u \in V(q)$, line 3 first generates a complete candidate vertex set $C(u)$ defined in Definition 4, and builds an auxiliary data structure $\mathcal{A}$ that maintains edges between candidate vertex sets. Based on candidate vertex sets and auxiliary data structures, line 4 generates a matching order defined in Definition 5. Line 5 recursively enumerates all results. M records mappings from query vertices to data vertices. When each element in $V(q)$ can find a valid mapping in the auxiliary structure a match has found, store that map in M and remove those vertices that got mapped from the auxiliary structure. The enumeration stops when a $C(u)$ becomes empty.

Definition 4. *Complete Candidate Vertex Set : Given q and G, a complete candidate vertex set $C(u)$ of $u \in V(q)$ is a set of data vertices such that for each $v \in V(G)$, if (u,v) exists in a match from q to G, then v belongs to $C(u)$.*

Definition 5. *Matching Order : A matching order ϕ is a permutation of $V(q)$. $\phi[i]$ is the ith vertex in ϕ.*

2.2 Filtering methods

In this thesis, we study 6 different filtering methods. Each of them builds an Auxiliary data structure based on their own pruning strategy. LDF and NLF are baseline filters, GQL,CFL, TSO and DP-iso utilized them.

LDF(Label and degree filter). LDF is the simplest filter we study. For vertices $v \in G$ and $u \in q$, if v and u have same label $L(v) = L(u)$ and degree $d(v) \geq d(u)$, then add v into $u's$ candidate set $C(u)$. $C(u) = \{v \in V(G) | L(v) = L(u) \wedge d(v) \geq d(u)\}$. LDF is an offline filter.

NLF(Neighbor label frequency filtering). NLF is built on top of LDF, besides the label and degree check, it has an additional neighbor label frequency check. Like LDF, NLF is an offline filter, the pruned vertex will not affect the pruning of other vertices. A label frequency $N(u, l)$ is the number of the neighbor vertex for vertex u that have label l. For e.g in Figure 1 $N(q1, A) = 1$ because $q1$ has only 1 neighbor having label A. For a $v \in C(u)$, the neighbor frequency for all labels in the neighborhood of v, $l \in L(N(u))$, $|N(v, l)| \geq |N(u, l)|$. For e.g $v1$ will not be prune by LDF from $C(q2)$ because $L(q2) = L(v1) \wedge d(q2) = d(v1)$, but will be prune by NLF because $|N(q2, B)| > |N(v1, B)|$ NLF computes $C(u)$ for $u \in q$ based on VertexID of u in ascending order.

GraphQL[9] : GQL is a state-of-the-art filter, unlike LDF and NLF, GQL is an online filter that utilizes the pruning information in the previous iteration to further reduce candidate set size. There are two stages in GQL, local pruning, and global refinement. Local pruning is offline pruning based on the profile of vertices. A profile is the lexicographic order of labels of u and neighbors within r hops of neighbors from u (for r=1 the local pruning is equivalent to NLF). For example in figure 1, the profile of $q2 = B$. The global refinement is an online filtering process that prunes candidate vertex sets generated by the local pruning with a pseudo subgraph isomorphism algorithm[8] as follows : Given $v \in C(u)$, build a bipartite graph B_v^u between $N(u)$ and $N(v)$. for $u' \in N(u)$) and $v' \in N(v)$ add an edge $e(u', v')$ to B_v^u if $v' \in C(u')$. So add an edge between u' and v' if v' is in u' candidate vertex set. check whether there is a semi-perfect matching in B_v^u, semi-perfect matching means all vertices in $N(u)$ are matched if not, remove v from $C(u)$.

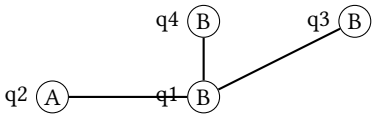

Figure 1 : GQL Example Query q

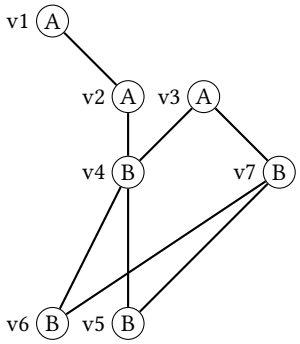

Figure 2 : GQL Example Graph G

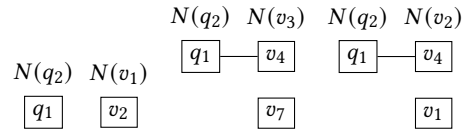

Figure 3 : GQL global refinement

Here I will present a running example of filtering in GQL. Given query graph q in figure 1, data graph G in figure 2. After local pruning $C(q_2) = \{v_1, v_2, v_3\}$. Then we further prune $C(q_2)$ with global refinement. After global refinement $C(q_2) = \{v_2, v_3\}$. v_2 got pruned from $C(q2)$ because $v2 \notin C(q1)$. $C(q2) = \{v_4, v_7\}$. $v2$ and $v3$ got kept in $C(q_2)$ because all vertex in $N(q_2)$ can find a match in $N(v_2)$ or $N(v_3)$ see figure 3. In global refinement, pruning is inline, the pruning of $C(u)$ potentially can affect the pruning of $C(u')$.

CFL [4]. Generation Rule : Given $X \in N(u)$ where $u \in V(q)$, $C(u)$ can be generated by $\Cap_{u' \in X} N(C(u'))$. Notice the union is only for neighbors got generated $C(u)$ in the BFS tree. Initially the $C(u_0)$ is generated by NLF.

Definition 6. *Filtering Rule : Given $X \in N(u)$ and $v \in C(u)$ where $u \in V(q)$, if there exists $u' \in X$ such that $C(u') \cap N(v) = \emptyset$, then v can be safely remove from $C(u)$ without breaking its completeness.[17]*

CFL first obtains a BFS tree q_t of q, and then constructs $\mathcal{A}$ in two phases :(1) Generate $C(u)$ along q_t level-by-level from top to bottom based on the Generation rule. (2) Refine $C(u)$ along q_t in a bottom-up order based on the Filtering Rule. Given u and its parent $u.p$ in q_t, A in CFL maintains edges between candidates in $C(u.p)$ and those in $C(u)$. The Generating phase is offline pruning, and the refinement phase is online pruning.

The following example illustrates the filtering in CFL. First, we generate a BFS tree q_t in the traverse order (q1 -> q2 -> q3 -> q4). $C(q1) = v_4, v_7$ generated by NLF. Then generate $C(q2)$ based on $C(q1)$ by find $N(C(q1))$. $C(q2) = v_2, v_3$ (the neighbor of q2 is

q3 and q1, we don't consider q3 because C(q3) is not generated yet). Then generated $C(q3)$ based on $C(q1)$ and $C(q2)$, $C(q3) = N(C(q1)) \Cap N(C(q2))$. $C(q3) = \{v_5\}$ Note that all v in C(u) for $u \in q$ must pass NLF. After generating $C(q3)$ we can do backward pruning on $q(2)$. v_2 will be pruned from $C(q2)$ because $C(q3) \Cap N(v2) = \emptyset$, $C(q2)$ become v_3. And continue to generate $C(q4)$ by intersecting with the neighbors' candidate set. Figure 6 shows the final $\mathcal{A}$. In this example, no candidate has been pruned in the refinement phase.

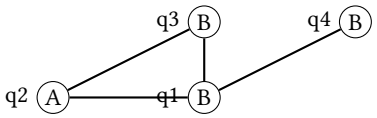

Figure 4 : CFL Example Query q

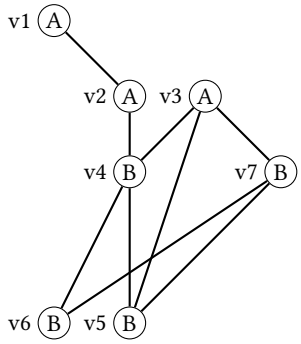

Figure 5 : CFL Example Graph G

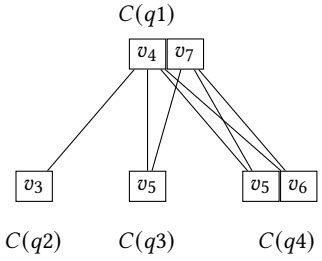

Figure 6 : CFL $\mathcal{A}$

DPiso DP-iso first performs a BFS on the query graph q from a randomly selected vertex, the BFS traversal order is denoted by δ. The initial $C(u)$ for $u \in q$ will be generated by NLF. Then DP-iso refines each $C(u)$ based on Filtering Rule refined in definition 6. Each refinement iteration contains a forward refinement backward refinement. In the forward refinement, DP-iso refines each $C(u)$ in reverse order of δ based on $C(u')$ where $u' \in N^{\delta}_{-}(u)$, which means replace $N(u)$ with $u' \in N^{\delta}_{-}(u)$ in Filtering Rule in definition 6. ($N^{\delta}_{-}(u)$ is the neighbors of vertex that positioned after u in δ.). In the backward refinement, DP-iso does the refinement along the order of δ based on $C(u')$ where $u' \in N^{\delta}_{-}$.

Example of filtering in DP-iso. Here we provide a running example of DPiso pruning for query graph q in figure7 and data graph G in 8. First, we generate the traversal order δ of q by using B. $\delta = (q1, q2, q3, q4)$, then generate the initial $C(u)$ for $u \in q$.

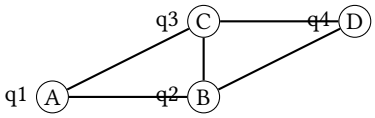

Figure 7 : DPiso Example Query q

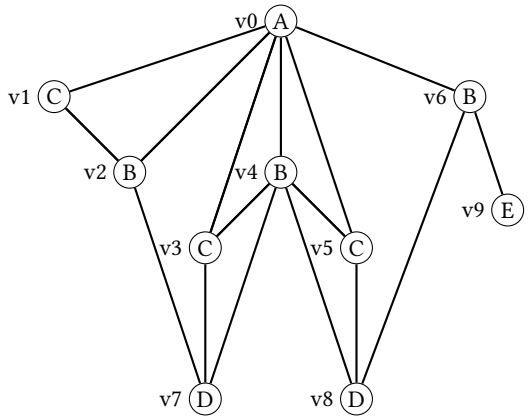

Figure 8 : DPiso Example Graph G

Figure 9 shows the initial $\mathcal{A}$. Then we do the forward refinement starting from q4, q4 is the last element in the traversal order and therefore doesn't have forward neighbors in $N^{\delta}_{-}(q3)$, and we continue to refine q3. The $N^{\delta}_{-}(q3) = q4$. $v1$ got pruned by Filtering Rule(definition6) because $N(v_1) \Cap C(q4) = \emptyset$. v_3, v_3 don't get pruned because there is intersection between $N(v_3), N(v_5)$ and $C(q_4)$, $C(q3) = \{v3, v5\}$. Then we continue to refine $C(q2)$. $N^{\delta}_{-}(q2) = q3, q4$. $v2$ and $v6$ got pruned from $C(u1)$ because $N(v2) \Cap C(q3) = \emptyset$,and $N(v6) \Cap C(q3) = \emptyset$. $v2$ is kept in $C(q2)$ because $N(v4) \Cap C(q3) = \{v_3, v_5\}$, and $N(v4) \Cap C(q4) = \{v_7, v_7\}$. Then we refine the last candidate set $C(q1)$, nothing got pruned. After completing the forward refinement we start the backward refinement. The idea is the same but instead, the refine along traversal order instead of reverse traversal order and refine based on $C(u')$ where $u' \in N^{\delta}_{+}(u)(N^{\delta}_{+}(u)$ is the neighbors of u position before u in δ). Figure 10 shows the final $\mathcal{A}$ after one refinement phase. DP-iso repeat k refinement phases, k is set by the user. The original paper sets k to 3. DP-iso records edges between candidates after refinement in $\mathcal{A}$.

2.3 Ordering method

The ordering method decides the order of mapping $u \in q$ to $q \in G$. The matching order can significantly improve the enumeration time by reducing the search space. The same candidate space with different matching orders can lead to significant differences in enumeration time. There are many methods to determine matching order based on various heuristics, we use the ordering method of GQL for our experiment in this thesis, which has been proven to be the most efficient in general experimentally[17]. The ordering method of GQL is based on the size of candidate vertex sets. GQL first selects $u^* = argmin_{u \in V(q)}|C(u)|$ as the start vertex of the

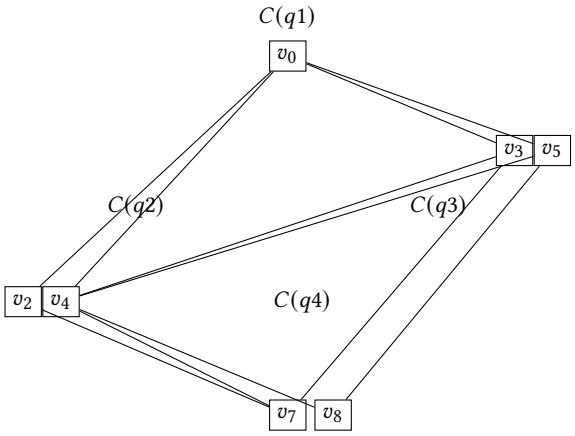

Figure 9 : Initial $\mathcal{A}$ generated by NLF

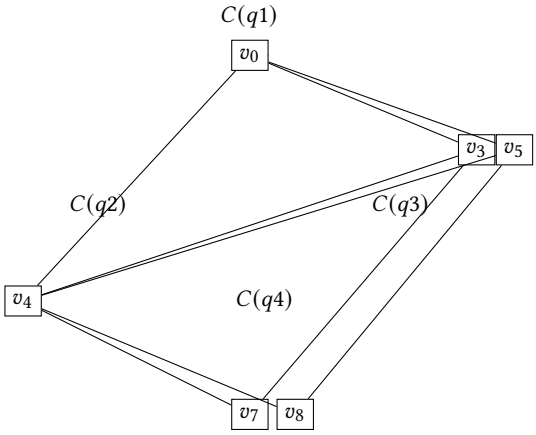

Figure 10 : $\mathcal{A}$ after refinement process

matching order ϕ. After that, GraphQL iteratively selects $u^* = argmin_{u \in N(\phi)-\phi}|C(u)|$ as the next vertex in ϕ[9].

2.4 Enumeration method

After generating $C(u)$ for $u \in q$ and generating matching order ϕ we come to the enumeration stage.

The enumeration method will recursively iterate through auxiliary data structure $\mathcal{A}$ and map $u \in q$ to a vertex in $C(u)$ along the matching order ϕ, return a match when all u got a valid match. Here is a high-level description of the enumeration method. We first choose an extendable vertex based on ϕ and M. An extendable vertex is query vertices u such that each $u' \in N_+^{\phi}(u)$ has been mapped in M but u has not. The first extendable vertex will be $\phi[1]$ since $N_+^{\phi}(\phi[1]) = \emptyset$. Then we call the enumeration method recursively on each local candidate and return M if all $u \in q$ have successfully found a valid mapping in the data graph. In this thesis, we use an enumeration method with set intersection-based local candidate computation, which has been proven to be the most efficient in general[17]. Algorithm 2 shows how the local candidates are generated. $\mathcal{A}_u^{u.p}(M[u.p]) = N(M(u.p)) \cap C(u)$.

Algorithm 2 Set intersection-based Local Candidate computation

1: **if** i = 1 **then** return $C(u)$
2: **if** $|N_+^{\phi}| = 1$ **then** return $\mathcal{A}_u^{u.p}(M[u.p])$
3: return $\bigcap_{u' \in N_+^{\phi}(u)} \mathcal{A}_u^{u'}(M[u'])$

3 SPECTRAL PRUNING STRATEGY

In this thesis, the out goal is to enhance the performance of preprocess-enumerate-based algorithms by reducing the size of the auxiliary data structure $\mathcal{A}$ and achieving a smaller search space for the enumeration stage. To achieve this goal, we improve the local candidate pruning in the filtering method with a new spectral pruning strategy. The local pruning employed in the filtering method of the algorithms we study in this thesis is LDF and NLF. We first generate $C(u)$ for $u \in q$ with the original local pruning. Then on top of those $C(u)$ we apply additional spectral pruning to further reduce the candidate space. A spectral pruning strategy provides additional pruning power by utilizing the neighborhood structure information in the graph.

The spectrum of a graph refers to the set of eigenvalues of its adjacency matrix or laplacian matrix. These eigenvalues carry essential information about the graph's structural properties. West et al. have demonstrated that by examining the eigenvalues of the adjacency matrices of two graphs g and G, we can determine whether graph g is Subgraph isomorphism to graph G. The spectral pruning strategy utilizes this property to eliminate false positive vertices from the candidate vertex space for each vertex $v \in q$.

The key idea behind the spectral filtering strategy is that if the query graph q is a subgraph isomorphism to a data graph G, each $u \in q$ must be able to find an embedding vertex $v \in G$ that the local structure around v and u are similar. For e.g if u has 5 neighbors v must have 5 neighbors as well for it to be a valid embedding of u. To utilize the local structure-preserving property for candidate pruning, the most straightforward way is to compare the local structure between pair of vertices (u,v) where $u \in q$, and $v \in G$ by performing two BFS of depth k where u and v are the roots, and compare the two traversing tree. However structured comparisons are computationally.

Definition 7. ***Vertex Topology Signature.*** *Given a graph G and a vertex $v \in G$, the adjacency matrix of $LNPT(G, v, n)$ is denoted as M. Let the eigenvalues of M be $\lambda_1 \leq \lambda_2 \leq ... \leq \lambda_m$. The Vertex Topology Signature of the vertex v is defined as the sorted list $topS(v) = [\lambda_1, ..., \lambda_t]$, where t is a specified parameter such that $t \leq m$ [20].*

Instead of doing a structured comparison, we can first represent the local structure around a vertex as a tree and then encode the information related to the tree to a numerical space, called vertex topology signature.[20] The Spectral Pruning Strategy will be pruning based on these signatures. The tree used to represent the local structure for vertices is called **Level-n Path Tree** and denoted as $LNPT(G, v, n)$. $LNPT(G, v, n)$ consists of all n-step simple paths from v to G. A simple path is a path in a graph with no repeated vertices. Figure 11 is an example of LNPT. Algorithm 3 shows the generation process of LNPT. By utilizing vertex topology signature,

the cost of checking can be significantly reduced by employing numerical comparison instead of structural comparison.

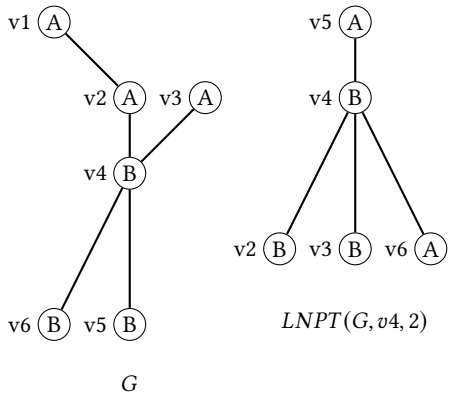

Figure 11 : Level-n Path Tree

The paper by West et al. has proven that for a given query graph q where $u \in q$ and a data graph G where $v \in G$, a vertex $v \in C(u)$ only if $topS(u) \leq topS(v)$. Here, $topS(v)$ represents the topological signature of v, which is the topS eigenvalue of the adjacency matrix of $LNPT(G, v, n)$, and $topS(u)$ represents the topological signature of the query vertex u. Specifically, $topS(v) = [\lambda_1, ..., \lambda_s]$ and $topS(u) = [\beta_1, ..., \beta_s]$, where $\lambda_i \leq \beta_i$ for $i = 1, ..., s$. This means that the first s eigenvalues of the adjacency matrix of $LNPT(q, u, n)$ will always be less than or equal to the first s eigenvalues of the adjacency matrix of $LNPT(G, v, n)$ if $v \in C(u)$.

Algorithm 3 Extracting Level-n Path Sub-tree(LNPT)

```
1:  Input : a graph G and a vertex v in G
2:  Output : level-n path sub-tree of depth n where vertex v is the
    root.
3:  Set the vertex v as the root of LNPT(v)
4:  Set the vertex set Visited = v
5:  for each r in N(v) do
6:      Insert the vertex r as child of v in LNPT(v)
7:      Insert the vertex r into the set Visited
8:      Search(r, n)
9:  procedure Search(v, n)
10:     if n==0 then return
11:     for each r ∈ N(v) do
12:         if r ∈ Visited then return
13:         Insert the vertex r as a child of v in LNPT(v).
14:         Insert the vertex r into the set Visited
15:         Search(r, n − 1)
16:         Delete the vertex r from the set Visited
```

However, computing eigenvalues is computationally demanding, as most eigenvalue computation algorithms have a time complexity of $O(n^3)$, where n is the number of rows in the adjacency matrix. Comparing all eigenvalues can lead to poor overall algorithm performance. To address this, a trade-off is made by focusing on the top 10 largest eigenvalues. Spectral graph theory suggests that a few of the largest eigenvalues play a significant role in determining the graph's topological structure[20]. Although this approach sacrifices some pruning power, the computation process for eigenvalues becomes considerably faster.

$$L_{ij} = \begin{cases} \deg(v_i), & \text{if } i = j, \\ -w_{ij}, & \text{if } i \neq j \text{ and } (i, j) \text{ is an edge in } G, \\ 0, & \text{otherwise.} \end{cases}$$

Figure 12 : Laplacian Matrix

By utilizing the Vertex Topology Signature we develop the pruning rule for our spectral filter.

$$v \in C(u) \text{ if } topS(v) > topS(u)$$

We precompute $topS(v)$ for each vertex v in the data graph before the query and store them in a file because it's a one-time overhead. We have to compute the signature for vertices in the query graph in run-time because each query graph is different. After the baseline local pruning(LDF/NLF) we check each $v \in C(u)$ whether they have satisfied the pruning rule $topS(v) > topS(u)$. Algorithm 4 show the pseudocode for the spectral filter.

Algorithm 4 Spectral Filter

```
1:  procedure Spectral Filter(G, q, s)
2:      Initilize C(u) by LDF/NLF
        SignatureData ← calculateVertexSignature(q, C(u), s);
        SignatureQuery ← LoadVertexSignature(G);
3:      for each u ∈ q do
4:          for each v ∈ C(u) do
5:              if SignatureData[v] ≤ SignatureQuery[u] then
6:                  Remove v from C(u)
7:      return C(u) for u ∈ q
8:  procedure calculateVertexSignature(q, C(u), s)
9:      Initilize EigenMatrix(u) = [], for all u ∈ q
10:     for each u ∈ q do
11:         Adj ← getAdjacencyMatrix(q, u, s)
12:         Tops ← getTopsEigenValue(Adj)
13:         EigenMatrix[v] ← Tops
14:         return EigenMatrix(u)
15: procedure getAdjacencyMatrix(q, u, s)
16:     return Construct an adjacency matrix for the level-n path
    tree LNPT(q,u,s).
```

A later study[12] shows that using the eigenvalue of the Laplacian matrix for Level-N Spanning Graph(LNSG) as Vertex Topology Signature is better than the Adjacency matrix + LNPT signature. Figure 12 shows how Laplacian Matrix is computed. Figure 13 shows an example of LNSG and compares it with LNPT. Algorithm 5. Shows the generation process of LNSG.

While both LNSG and LNPT are a tree-like structure that is used to represent the local structure around a vertex. One of the significant advantages of LNSG (Level-n Simple Path Graph) is its ability to generate signatures faster. In comparison to LNPT (Level-n Path Tree), which represents the topological structure using subtrees

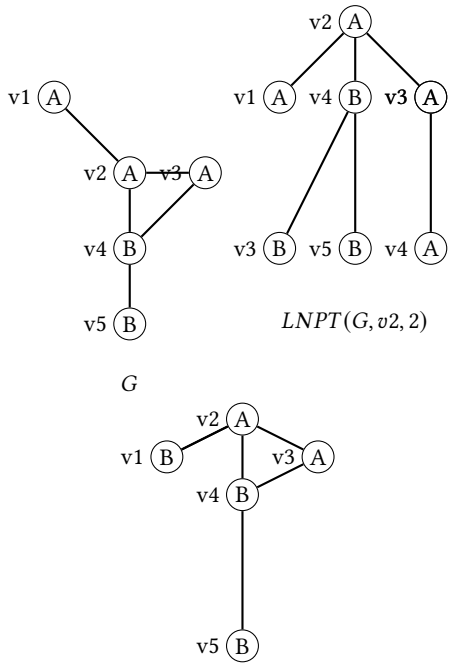

Figure 13 : LNPT vs LNSG

Algorithm 5 Extracting Level N spanning graph(LNSG)

```
1: Input : a graph G and a vertex v in G
2: Output : level-n spanning graph depth n where vertex v is the root.
3: Set the vertex LNSG_v = {v}
4: Set the edge set LNSG_e = ∅
5: Search(r, n)
6: for each edge e ∈ G do
7:     if two vertices of edge e exist in LNSG_v then Insert the edge e into LNSG_e
8: procedure Search(v, n)
9:     if n==0 then return
10:    for each r ∈ N(v) do
11:        if r ∉ LNSG_v then
12:            Insert the vertex r into LNSG_v
13:            Search(r, n − 1)
```

containing all n-step simple paths from a vertex, LNSG eliminates the presence of duplicate vertices. This distinction contributes to improved efficiency, particularly in eigenvalue computation. As a result, LNSG achieves faster eigenvalue computation leading to improved efficiency compared to LNPT overall.[12].

The advantage of using the Laplacian matrix is its stronger pruning power compared to the adjacency matrix. Although the study did not theoretically prove that the signatures generated by the Laplacian matrix carry more information, experiments have shown that the signature generated by Laplacian Matrix provides superior pruning power compared to the signature by the Adjacency matrix. [12].

In summary, LNSG offers faster signature generation than LNPT. Compute the signature using the laplacian matrix of LNPT have stronger pruning power than the signature computed by the adjacency matrix. Therefore we compute the vertex signature using LNSG and its Laplacian matrix in our spectral filter to further reduce $\mathcal{A}$ size. In the actual implementation, we lose the constraint in Algorithm 5 line 6-7 for better performance in exchange for a small lost of information in the vertex signature.

3.1 Indexing Vertex Topology Signature and Overhead

To enhance the efficiency of the pruning process, we precompute the Vertex Topology Signature for each vertex in the data graph and store them as an index. During pruning, we simply perform a linear scan of the index for each vertex $u \in q$. For those vertices $v \in G$ that have a Vertex Topology Signature $tops(v)$ greater than or equal to $tops(u)$, we add v to $C(u)$. The signature for vertices in the query graph is computed at runtime. Typically, the query graph size is much smaller than the data graph, making it acceptable to compute the signatures for the query graph dynamically, unless the query graph size approaches that of the data graph. The data graph index does not need to be recomputed when inserting or removing vertices. We only need to update the signatures for vertices reachable within n hops from the affected node. Thus, this method can support dynamic data graphs.

Time complexity : The time complexity for the spectral pruning process is given by $O(|q| \cdot s \cdot |LNPT(q, u, n)|^2) + O(|q| \cdot |G|)$. The first term represents the time required to compute the topology signature for each vertex in q, while the second term corresponds to the time taken to iterate through the index, with each vertex in q being iterated once. In Section 4, we will conduct experiments on datasets with various properties to observe the impact of spectral pruning on overall performance.

Space complexity : The space complexity is determined as $O(|V(q)| \cdot 10) + O(|V(G)| \cdot 10)$. The first term represents the storage required for the signatures of vertices in the query graph, and the second term represents the storage for the signatures of vertices in the data graph.

One important point to note is that the order of applying local pruning strategies does not matter, as they solely prune false positive vertices based on the information from both the query graph and the data graph. Local pruning strategies do not consider the potential impact of candidate vertex sets on each other, which is offline pruning. Thus, pruning a vertex with the Local Degree Filtering (LDF) method, for example, does not affect the pruning decisions made by the Neighborhood Label Filtering (NLF) or spectral pruning methods. However, in the global refinement stage, the pruning of the candidate vertex set is online. Pruning a vertex in $C(u)$ can subsequently lead to the pruning of another vertex in $C(e)$, and so on, in a cascading manner. Consequently, the effectiveness of local pruning strategies has a significant impact on the size of the final auxiliary structure.

4 EXPERIMENTS

In this section, we present our experimental evaluation of the spectral pruning strategy on various datasets with different data graph properties. Our experiments are conducted based on the code framework provided by Sun in his survey [17]. As described in Section 2, Sun has abstracted the preprocessing-enumeration-based subgraph matching algorithm into three components : the Filtering Method, Ordering Method, and Enumeration Method. He has implemented a flexible code framework that allows the combination of different algorithm components, enabling the study of the individual impact of each component on the final algorithm performance.

We leverage this framework to implement the spectral pruning filter on top of two local pruning filters. Our experiments compare the pruning effectiveness and overall performance between algorithms with the original filtering method and those with the spectral filter enhancement. To specifically examine the filtering method, we keep the ordering method and enumeration method fixed across different filters. The ordering method utilizes GQL, which refers to the GQL algorithm from GraphQL. The enumeration method employs a set-intersection-based local candidates computation method. These methods are recommended in Sun's[17] survey as they generally demonstrate superior performance.

Goal of comparisons In the following experiment section, we conducted three major experiments : 1. Pruning power experiment. This experiment aims to compare the difference in candidate vertex set size with and without the additional spectral filter. 2. Performance experiment. This experiment aims to investigate whether the improvement of pruning power leads to an improvement in query evaluation time improve. 3. Wildcard label. This experiment aims to study the performance of spectral filters under the scenario where label information is lacking.

4.1 Experiment Setup

Server Information : All algorithms were compiled in C++17. The experiments are conducted on an ubuntu 18LTS machine with Intel Core i9 10940X 3.3GHz 14-Core CPU and 256GB RAM.

Data Graphs : Eight real-world datasets used in previous work related to subgraph matching[[2],[6],[7],[9],[11],[15],[16],[18]] have been selected for our experimental analysis. Each dataset exhibits distinct characteristics, including varying average degrees and numbers of label types.(see table 2)

Query Graphs : For each dataset, The query graphs are generated with two properties : Dense and Sparse. Both dense and sparse queries are generated by a random walk within the data graph. Dense queries possess an average degree greater than 3, while sparse queries have an average degree lower than 3. Each query is available in three different sizes : 16, 24, and 32 vertices. However, for the Wordnet and Human datasets, the query sizes are adjusted to 12, 16, and 20 vertices. This adjustment is made due to the inherent challenges associated with these datasets. Human exhibits a high density, while most vertices in Wordnet share the same label, making them particularly demanding test cases.

Metrics : We employ three metrics to evaluate the effectiveness of the spectral filter :

(1) **Increase in pruning power :** This metric measures the improvement in pruning achieved by the spectral filter (how much less candidate after using the spectral filter). It is quantified as the difference between the sums of $|C(u)|$ for all $u \in q$.

(2) **Number of searched nodes in the enumeration process :** This metric indicates whether the search space within the auxiliary data structure has been reduced. A lower number of searched nodes suggests that the spectral filter has effectively pruned irrelevant candidates. The purpose of this measure is to isolate the actual search space from the overhead in the filtering process for computing vertex signatures.

(3) **Total time to find all matches :** This metric assesses the impact of the spectral filter on the overall performance of the subgraph matching algorithm. It measures the time required to identify all matches and serves as an indicator of the filter's contribution to improving efficiency.

Dataset	Name	$\|V\|$	$\|E\|$	$\|\Sigma\|$	d
WordNet	wn	79,853	120,399	5	3.1
Youtube	yt	1,132,890	2,987,624	25	5.3
DBLP	db	317,080	1,049,866	15	6.6
HPRD	ye	3,112	12,519	71	8.0
US Patents	up	3,774,768	16,518,947	20	8.8
eu2005	eu	862,664	16,138,468	40	37.4
Human	hu	4,674	86,282	44	36.9
Yeast	ye	3,112	12,519	71	8.0

Table 2 : Properties of real-world datasets

4.2 Pruning power experiment

In this section, we present a comprehensive experimental analysis of our spectral pruning strategy using six different filters across eight real-life datasets. We applied both enhanced and non-enhanced filters to each dataset and compared the resulting sum of candidate vertex sets, denoted as $C(u)$ for $u \in q$. The experimental findings are presented in Figures 14 to 21.

The experimental results demonstrate that our spectral pruning strategy improved the pruning power of all the filters employed in our experiment. However, the extent of improvement depends on the number of label types present in the data graph. In datasets with a large number of label types (e.g., HPRD with $|\Sigma| = 71$) in the data graph, the improvement in pruning power is minimal, except for the baseline method LDF, which shows a significant gain. This is because LDF solely relies on a simple label-matching strategy, checking if the candidate $v \in C(u)$ has the same label as u. Therefore its pruning power is very weak compared with other state-of-the-art filters.

Interestingly, in WordNet, the dataset where spectral pruning achieves the most significant improvement 14, not only the baseline method but also the state-of-the-art methods exhibit a substantial increase in pruning power. Another noteworthy observation is that the state-of-the-art filters perform similarly to the baseline method in WordNet, whereas in other datasets, they outperform

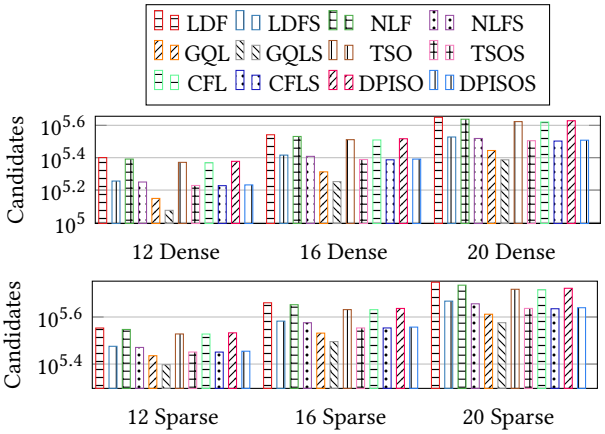

Figure 14 : Pruning Power Wordnet Queries

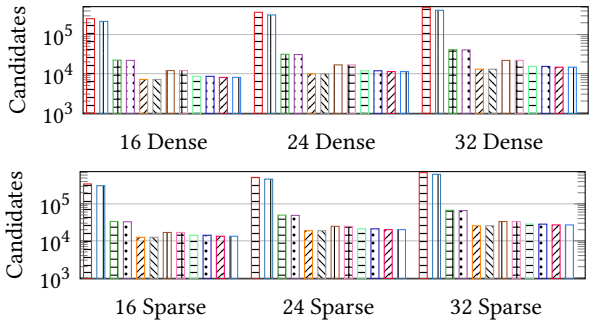

Figure 15 : Pruning Power Youtube Queries

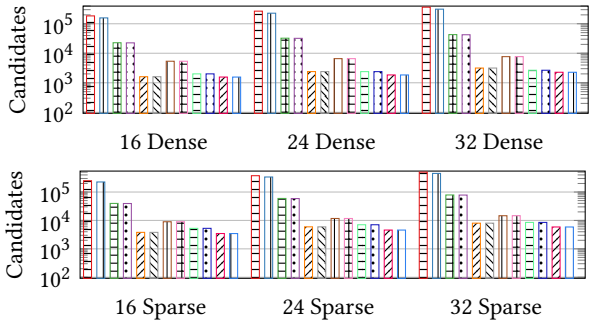

Figure 16 : Pruning Power DBLP Queries

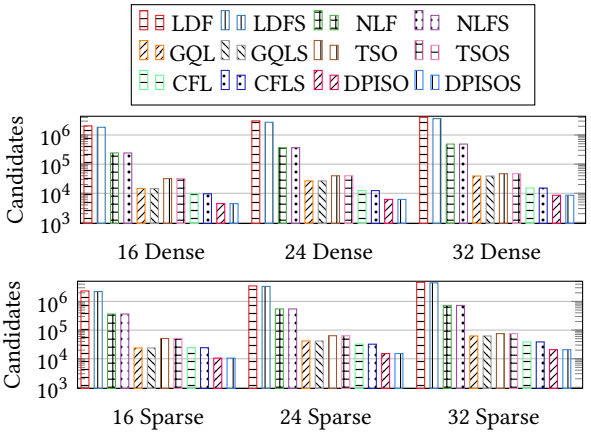

Figure 17 : Pruning Power US Patents Queries

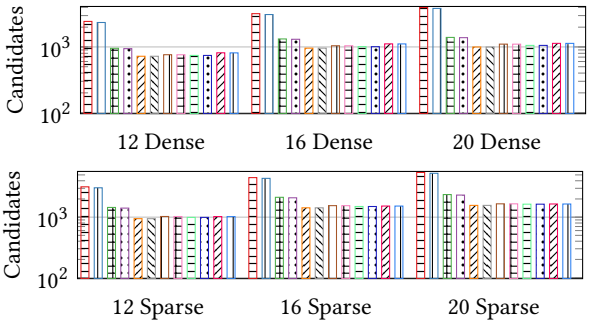

Figure 18 : Pruning Power Human Queries

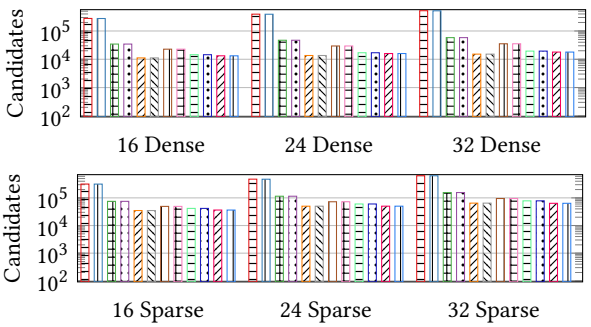

Figure 19 : Pruning Power eu2005 Queries

the baseline method by a significant margin. WordNet stands out due to its unique characteristics : it has only five labels, the label distribution follows a power-law pattern, and most vertices share the same label.

We hypothesize that the reason spectral pruning performs exceptionally well in WordNet is because of the power-law label distribution, most vertices in the data graph have the same label. This cripples the pruning strategies that heavily rely on exploiting label information, explaining why the state-of-the-art filters perform similarly to the baseline method in WordNet. As the label-based pruning strategies are ineffective, numerous false positive candidates remain unpruned, the spectral pruning strategy prunes these candidates that suppose to be pruned by the label-based pruning strategy, leading to its exceptional performance in WordNet. In the later experiment, we further validate our hypothesis by introducing the "wildcard" label type in the query graph.

In summary, this experiment demonstrates that the spectral pruning strategy improves the pruning power of various filters. The degree of improvement depends on the dataset's label characteristics, with WordNet showing the most significant pruning power increase due to its high presence of vertices with the same label. These findings provide valuable insights into the effectiveness of spectral pruning in optimizing query processing in real-life datasets.

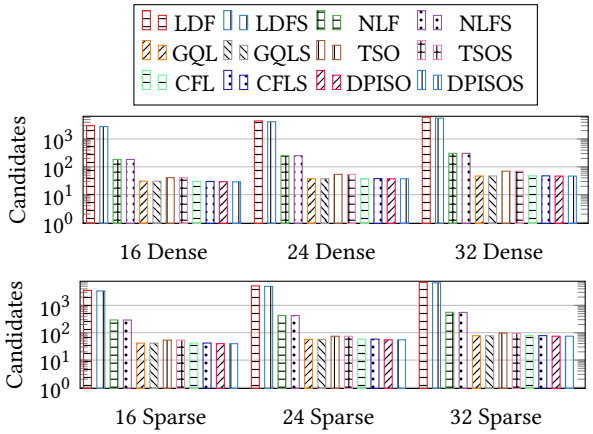

Figure 20 : Pruning Power HPRD Queries

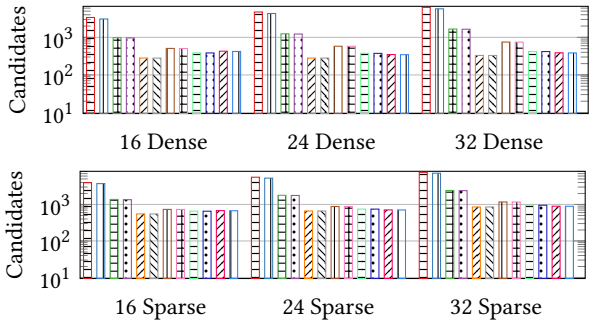

Figure 21 : Pruning Power Yeast Queries

4.3 Wildcard label

In Section 4.2, we hypothesized that spectral pruning performs well in WordNet due to the power law label distribution, which provides limited label information for label-based filters. This hypothesis suggests that the spectral filter's additional pruning power can eliminate candidates that cannot be pruned by the label-based filter. Consequently, we postulate that the spectral filter is most effective when label information in the data graph is lacking.

To validate our hypothesis, we need to diminish the label information in the data graphs where spectral pruning doesn't enhance(or very small gain) pruning power. One approach is to perform a surjective mapping of label sets to a smaller subset. For example, we can map labels "a" and "b" to label "a" in the graph, effectively reducing the variety of labels. However, this approach has a drawback as it reduces label variety. When merging labels "a" and "b" into the label "a," the label "b" disappears from the graph, resulting in a reduced label category count.

A better approach to cripple label-based pruning strategy is to introduce a wildcard label. The wildcard label can be treated as any label during the pruning process. By introducing a wildcard label, we can impair the label-based pruning strategy while preserving label variety. Although any existing label may still appear in the query, unlike the subjective mapping approach which eliminates the existence of certain labels. In our experiments, we represent the wildcard label using the number of label types, Wildcard label = $|\Sigma|$ (The label started by 0).

The experiment follows a similar setup to the pruning power experiment. However, for each query, we randomly assign a vertex in the query graph based on a given probability. When the probability is 0, the query remains identical to the original query, while a probability of 1 creates a query consisting solely of wildcard labels.

Implementing the wildcard label requires addressing the challenge that the wildcard label only exists in the data graph. The framework builds an index for efficient neighbor frequency lookup, storing the label ID as the key and the frequency of its appearance in the neighborhood as the value. However, this presents an issue when encountering the wildcard label, as it only exists in the query graph and not the data graph.

To handle this issue we modify the NLF as follows : In the neighbor frequency check, we do not consider the label frequency for the wildcard label. Instead, we use a counter to keep track of the number of wildcard labels in the neighborhood of a vertex. We check if all other neighbors pass the label frequency test while summing the frequencies of non-wildcard labels. There are two scenarios where a vertex fails the neighbor frequency test. First, if there is a label present in the vertex's label set but not in the neighbor's labels, the neighbor vertex is pruned as before. Second, if all neighbors pass the frequency test with non-wildcard labels, but the sum of label frequencies plus the wildcard label count is less than the sum of label frequencies in the vertex, it indicates the existence of a wildcard label, and regardless of its assigned value, would violate the neighbor frequency rule. For example, if $NLF(u)$ has $\{a:2, b:1, *:1\}$ and $NLF(v)$ has $\{a:2, b:1\}$, then v is not in the candidate set of $C(u)$. However, if $NLF(u)$ has $\{a:2, b:1, *:1\}$ and $NLF(v)$ has $\{a:2, b:2\}$, v will be the candidate of $C(u)$ because the wildcard label can be assigned to b without break NLF rule.

The remaining algorithm implementation is straightforward with the wildcard label, where we allow any $v \in G$ to pass the label check when a query vertex has the wildcard label. The most challenging part is the software engineering aspect where adapt the framework to accept the wildcard label. The algorithm itself did not change.

The experiment results are presented in Figure 22 and 23. We display the results in terms of both the percentage of candidate set size reduction and the exact number of reduced candidates. The results show that the LDF doesn't experience much change in pruning power as more wildcard labels are introduced in the query. This is because the label-based filter in LDF is relatively weak and only checks for identical labels between vertices, most of the pruning power comes from the degree filter. In the case of WordNet, all algorithms exhibit similar behavior to LDF. This is due to the data graph already having a power law label distribution, resulting in limited label information(most labels are the same).

Unlike other datasets where the pruning power for each algorithm has monotonic gain as the percentage of wildcard label increase in the query graph, WordNet has a valley-shaped result in this experiment. This is because in the original query, most labels are already identical since it generates by a randomwalk in a data graph has power law label distribution. Introduce some wildcard

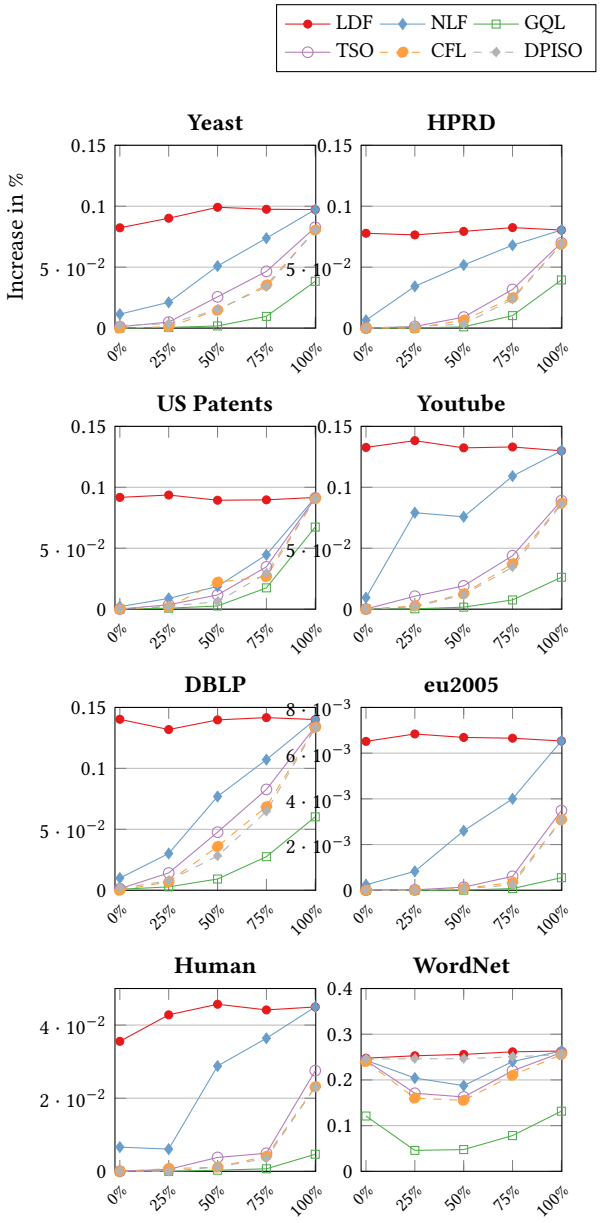

Figure 22 : Spectral Pruning vs wildcard labels %

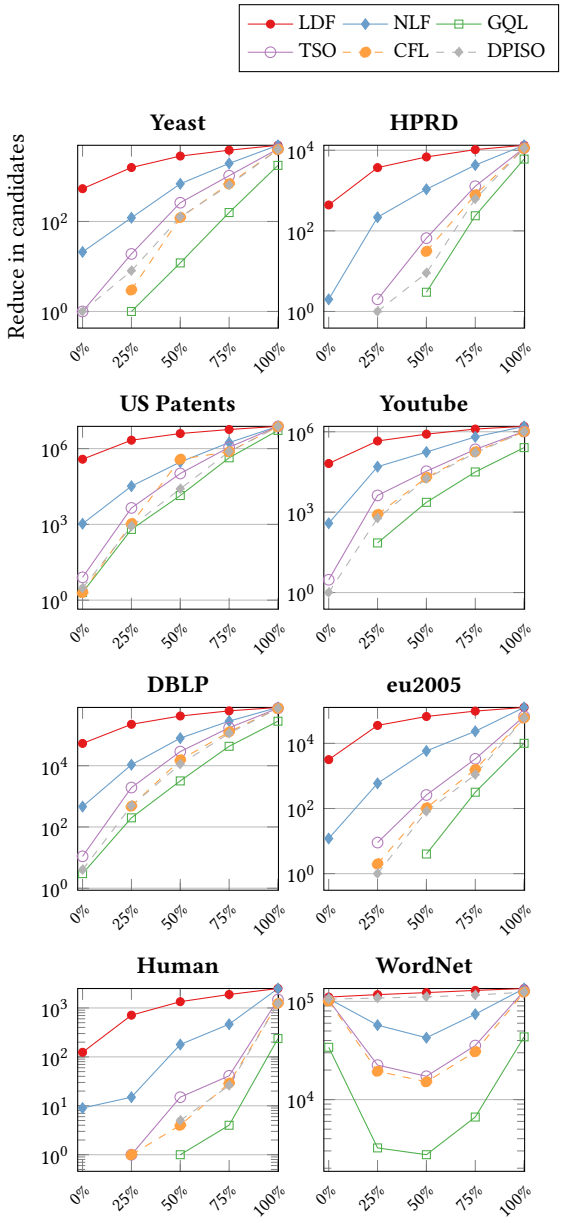

Figure 23 : Spectral Pruning vs Wildcard Labels the number of candidates reduced

labels activate some pruning power of the label-based filter therefore there is an increase in pruning power before the percentage of wildcard labels reaches 100%. When all vertex in the query becomes the wildcard query it's pruning power back to the initial position, since the label-based filter got crippled again.

Based on the experiment results, we have gained more confidence in our hypothesis that the spectral filter performs best in label-poor scenarios.

4.4 Performance experiment

In this section, we evaluate the performance of spectral pruning enhanced and original algorithms on different datasets. We compare the algorithms based on two metrics : query evaluation time and node visits during the enumeration process. To ensure a reasonable experiment duration, we set a 300-second timeout for each query and terminate the algorithm when it finds 100,000 matches. For those queries unresolved after time out we have padded the result as 300 seconds. The padding is very important because there might

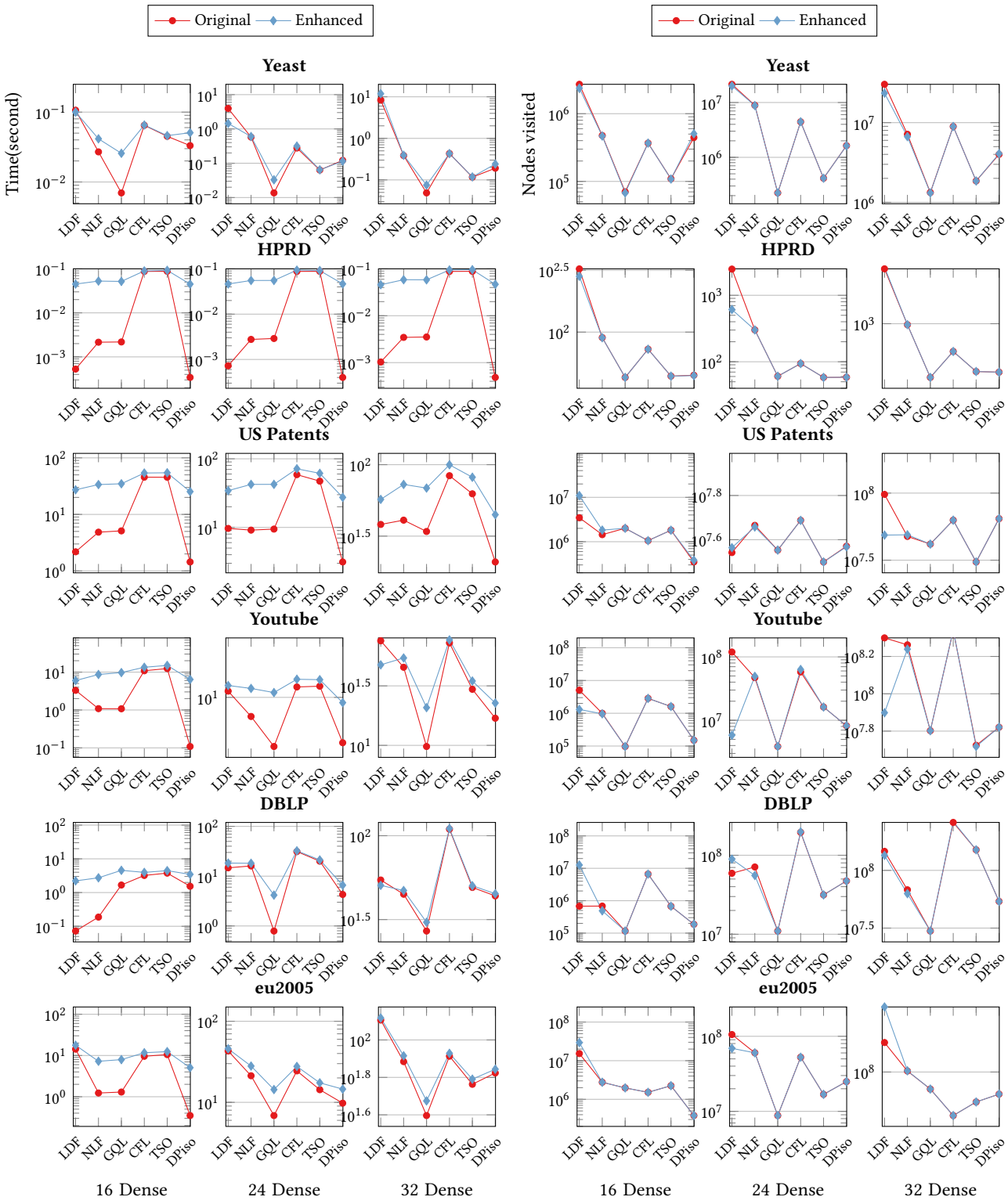

Figure 24 : Performance comparison time vs nodes visit.

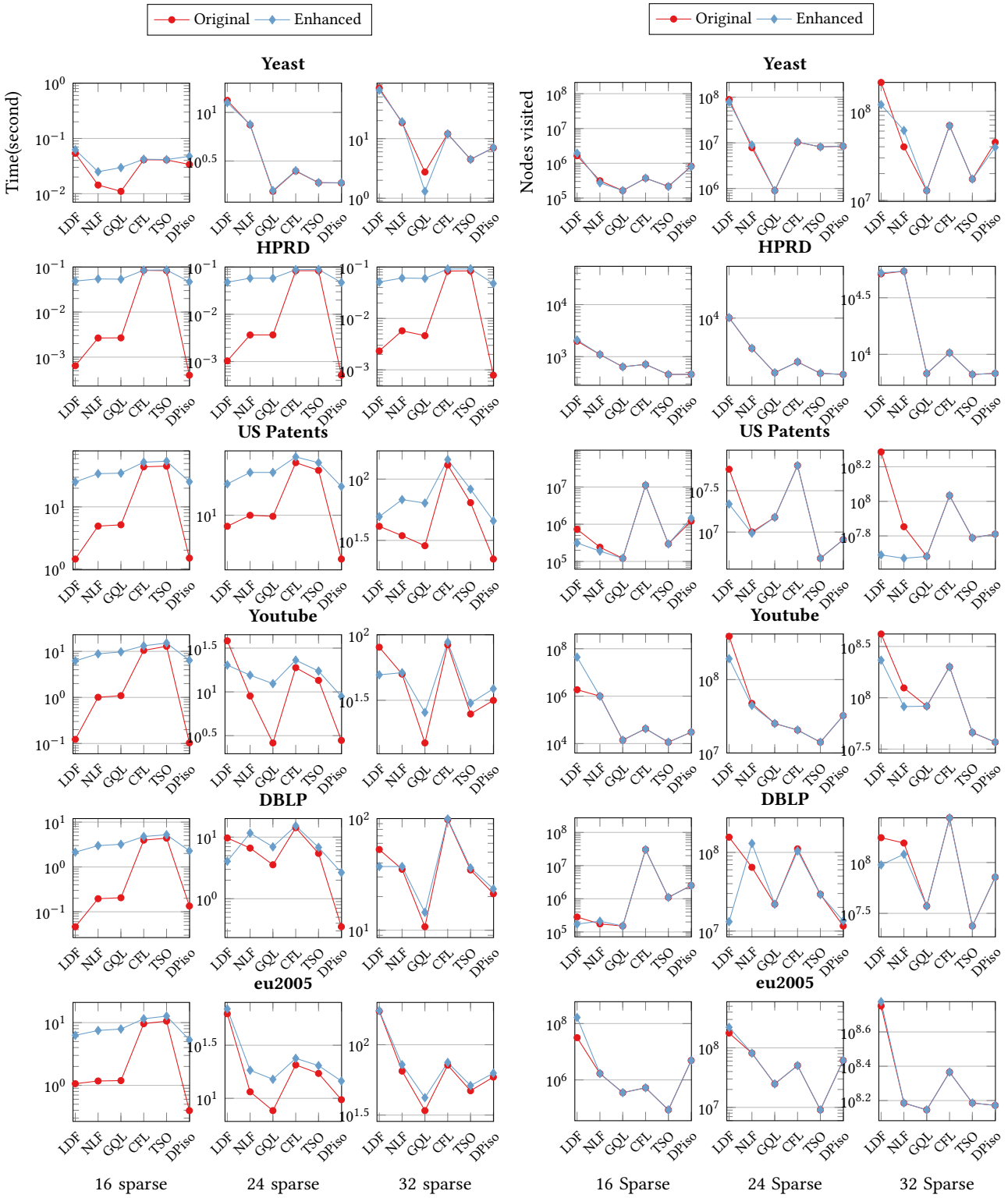

Figure 25 : Performance comparison time vs nodes visit.

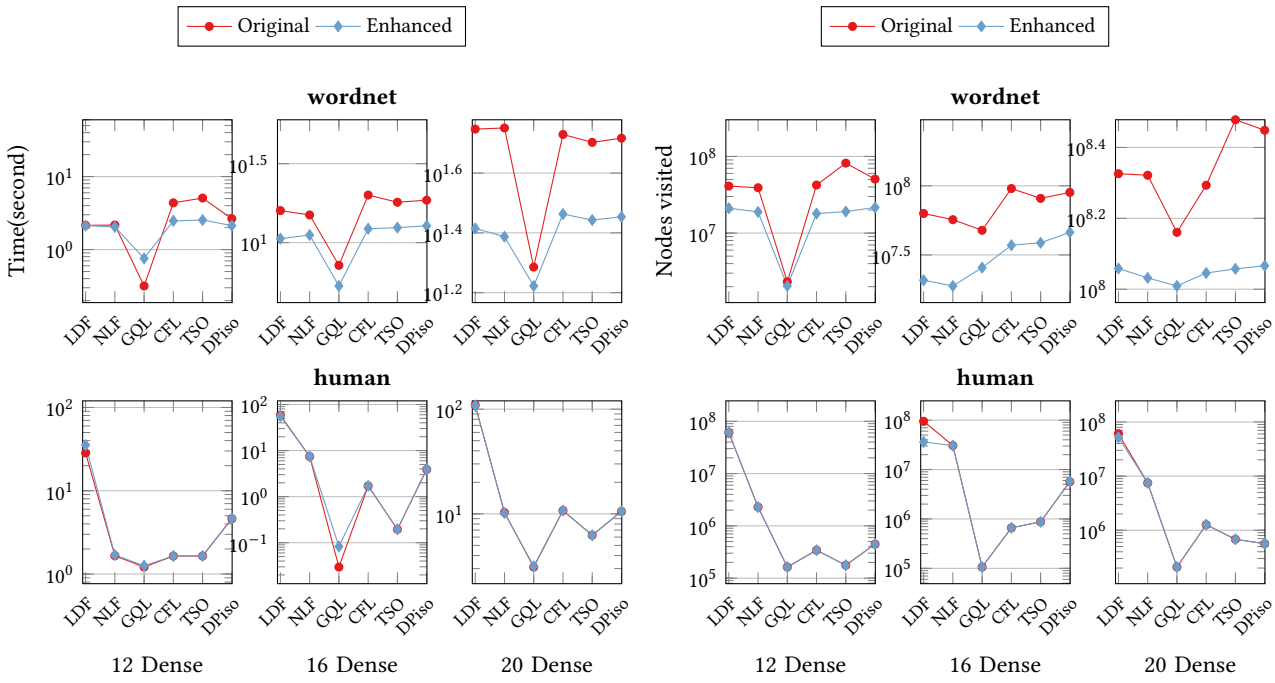

Figure 26 : Performance comparison time vs nodes visit.

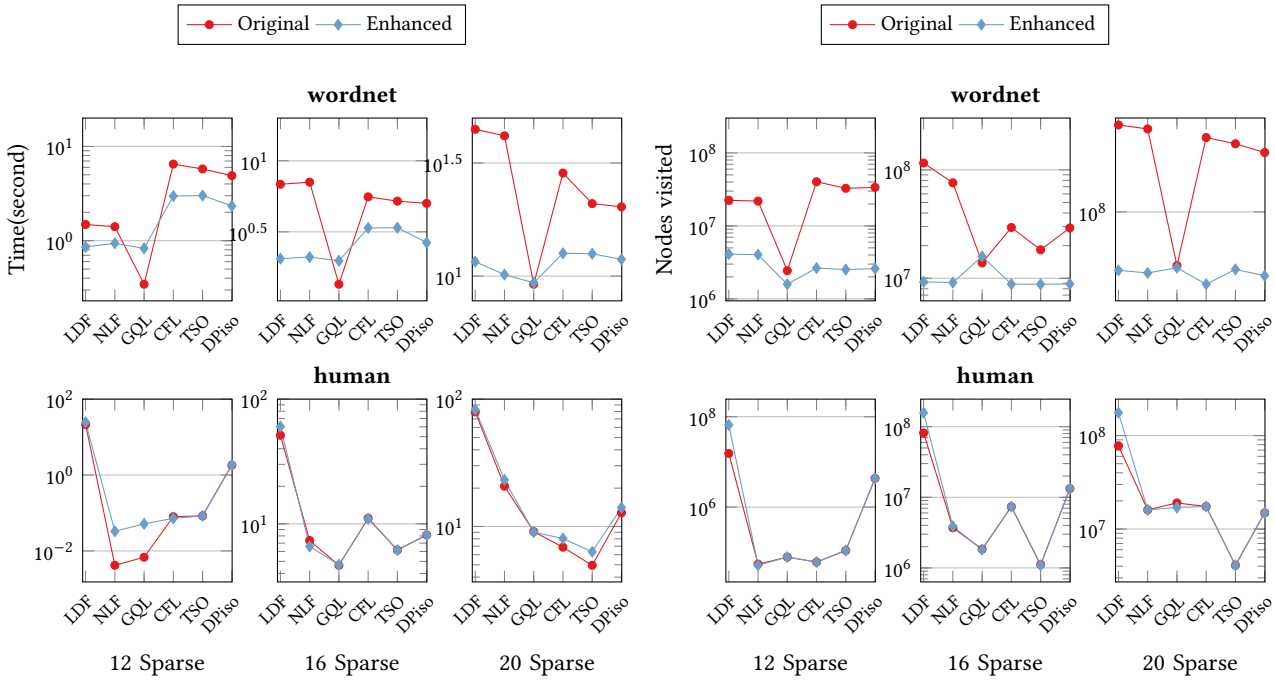

Figure 27 : Performance comparison time vs nodes visit.

be a scenario where a query can only be completed within time-out with the enhancement of the spectral filter. Without the padding, the slower filter will have faster performance because those queries they can't complete will be ignored.

We chose 100,000 as the match limit because the hardness of a query significantly impacts query time. Easy queries require only a few vertex traversals to determine if an instance is a match, while hard queries require significantly more traversals. If we set the match count too high, hard queries may not reach the match count within the time limit. Our focus is on observing the performance of spectral pruning in hard queries since easy queries can find all matches quickly, making the overhead of building the spectral index unnecessary. We expect spectral pruning to excel in scenarios where queries are challenging to evaluate and have a large search space (large auxiliary data structure), especially in queries with many vertices. By pruning candidates that cannot be pruned by other filters, spectral pruning reduces the search space using spatial information in the graph and saves valuable time.

Figure [24,25,27,27] illustrates the performance of spectral pruning compared to the original algorithms across different dataset sizes and properties. By observing the evaluation time, we notice that most datasets show worse performance in both dense and sparse queries, except for WordNet. This is expected, as only WordNet experiences a significant reduction in candidate set size. Notably, algorithms with LDF filters demonstrate better improvement due to their weakness in pruning power. Other filters have pruned candidate spaces that closely match the actual result, leaving fewer false positive vertices for the spectral filter to prune.

When considering the performance in terms of node visits, most algorithms show no significant difference after enhancing with spectral pruning, except for the baseline method, which exhibits significant improvement due to its weak pruning power. This implies that the reason algorithms take longer to execute a query is due to the overhead of spectral pruning.

WordNet stands out as the only dataset where all algorithms exhibit significant performance improvement after the enhancement with spectral filtering. This is particularly evident in queries with dense properties, where even the algorithm with the strongest filter (GQL) experiences a notable performance boost. However, in sparse queries, GQL remains dominant due to the limited structural information available in a sparse graph for the spectral filter to utilize in pruning.

One noteworthy observation is that the size of the candidate vertex set does not solely determine the search space's size. Even if a candidate set B is strictly smaller than set A, it can still require more node visits in the enumeration stage to reach the same number of matches. Initially, we speculated that this was due to the match limit. The matching order for a larger candidate set might accidentally be better for reaching the match limit. However, we discovered counterexamples in the experiment data, indicating that the smaller candidate set does not always perform better than a larger set even in exact evaluation. Hence, this phenomenon must be attributed to the change in the matching order. To validate this, we conducted an experiment by providing the query with a smaller candidate set as the matching order for the query with a larger candidate set. The results demonstrated that under the same matching order, the candidate set with a smaller size consistently outperforms the larger set in the enumeration stage by visiting fewer nodes, although the improvement is fewer than the matching order generated by a smaller candidate set on average. Further investigation into this phenomenon requires a deeper analysis of the matching order, which falls beyond the scope of this thesis. Our thesis focuses on reducing the candidate set size, and we leave this issue for future research.

5 CONCLUSION

In this thesis, We have demonstrated that the introduction of a spectral filter can enhance the pruning power of existing filters, particularly in scenarios where the label information is limited. This improvement in pruning power has consequently led to enhanced performance in preprocessing-enumeration-based graph matching algorithms.

We evaluated queries on datasets with varying properties and compared the pruning power of the original filters with the enhanced filters. We observed a significant improvement in pruning power when applying the additional spectral filter to the "wordnet" dataset, where most vertices share the same label. Furthermore, by introducing wildcard labels to the queries and evaluating different datasets with varying percentages of vertices replaced by wildcard labels, we demonstrated that the pruning power of the queries improved with an increasing percentage of wildcard labels. These experiments provided further evidence supporting the superiority of spectral filtering under the label-poor scenarios.

We compare the performance of the original and enhanced algorithms in terms of query evaluation time and the number of visited vertices during the enumeration phase. Among the eight datasets examined, only the "wordnet" dataset demonstrated a reduction in query evaluation time. However, across all datasets, there was a decrease in the number of visited nodes. This indicates that, except for the "wordnet" dataset, the overhead associated with spectral filtering did not yield significant improvements. Consequently, we only recommend utilizing the spectral filter specifically for solving subgraph-matching queries in scenario where label-information is limited for pruning.

RÉFÉRENCES

[1] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci : Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*. 1447–1462.

[2] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI : Compact Embedding Cluster Index for Scalable Subgraph Matching *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1447–1462.

[3] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products *(SIGMOD '16)*.

[5] S. Fortin. 1996. The graph isomorphism problem. Department of Computing Science, University of Alberta.

[6] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching : Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1429–1446.

[7] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso : Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases *(SIGMOD '13)*.

[8] Huahai He and Ambuj K. Singh. 2006. Closure-tree : An index structure for graph queries *(CIDE '08)*.

[9] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-Time : Query Language and Access Methods for Graph Databases *(SIGMOD '08)*.

[10] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile equivalences : Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data*. 925–937.

[11] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-Depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proc. VLDB Endow.* 6, 2 (dec 2012), 133–144.

[12] Qinbao Song Lei Zhu. 2011. A Study of Laplacian Spectra of Graph for Subgraph Queries.

[13] Ciaran McCreesh, Patrick Prosser, Christine Solnon, and James Trimble. 2018. When Subgraph Isomorphism is Really Hard, and Why This Matters for Graph Databases. 61, 1 (jan 2018), 723–759.

[14] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. 26, 10 (oct 2004), 1367–1372.

[15] Xuguang Ren and Junhu Wang. 2015. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. 8, 5 (jan 2015), 617–628.

[16] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness : An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.* 1, 1 (aug 2008), 364–375.

[17] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching : An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1083–1098.

[18] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.* 5, 9 (may 2012), 788–799.

[19] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (jan 1976), 31–42.

[20] Lei Zou, Lei Chen, Jeffrey Xu Yu, and Yansheng Lu. 2008. A Novel Spectral Coding in a Large Graph Database.