SUFF: Accelerating Subgraph Matching with Historical Data [Technical Report]

Xun Jian
Hong Kong University of Science and Technology
Hong Kong, China
xjian@connect.ust.hk

Zhiyuan Li
Hong Kong University of Science and Technology
Hong Kong, China
zlicw@cse.ust.hk

Lei Chen
Hong Kong University of Science and Technology
Hong Kong, China
leichen@cse.ust.hk

ABSTRACT

Subgraph matching is a fundamental problem in graph theory and has wide applications in areas like sociology, chemistry, and social networks. Due to its NP-hardness, the basic approach is a brute-force search over the whole search space. Some pruning strategies have been proposed to reduce the search space. However, they are either space-inefficient or based on assumptions that the graph has specific properties. In this paper, we propose SUFF, a general and powerful structure filtering framework, which can accelerate most of the existing approaches with slight modifications. Specifically, it builds a set of filters using matching results of past queries, and uses them to prune the search space for future queries. By fully utilizing the relationship between matches of two queries, it ensures that such pruning is sound. Furthermore, several optimizations are proposed to reduce the computation and space cost for building, storing, and using filters. Extensive experiments are conducted on multiple real-world data sets and representative existing approaches. The results show that SUFF can achieve up to 15X speedup with small overheads.

PVLDB Reference Format:
Xun Jian, Zhiyuan Li, and Lei Chen. SUFF: Accelerating Subgraph Matching with Historical Data [Technical Report]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

In this paper, we study subgraph matching, one of the fundamental problems in graph theory. Given a *data graph* d and a *query graph* q, the goal is to find all subgraphs of d which are isomorphic to q. Subgraph matching has wide applications in many research areas. For example, it is used in comparing large graphs in sociology, chemistry, telecommunication, and bioinformatics [26, 32]. It is also used to monitor potential terrorists by searching threat patterns in activity networks[6]. Nevertheless, it can be adapted to track the evolution of social networks [16] and to identify useful properties in recommendation networks [24].

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

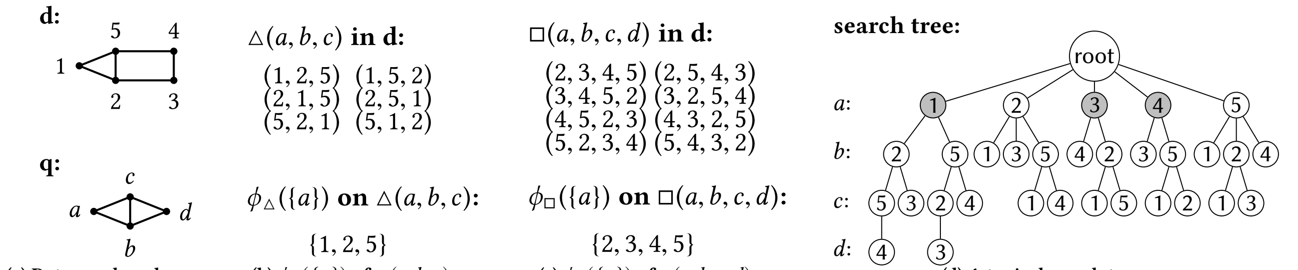
1.1 Motivation

Due to the NP-hardness of the subgraph matching problem [7], the basic approach to solve it is to explore the whole search space (i.e., to enumerate all subgraphs of d and then compare each of them to q) by DFS or BFS. Recent developments on such approach include [3, 4, 11, 14, 28, 29].

Different optimizations have been proposed in these approaches to reduce the search space. For example, vertex degree and label frequency are utilized to prune candidates and also to find a better matching order. The connectivity index and path index are constructed to refine candidate sets. More advanced pruning strategies like *failing set* are proposed to prune the search space. These optimizations are designed based on expert experience, artificial rules, or common patterns observed in real-world graphs, so are suitable for different situations. In a recent study [34], it is observed that each optimization method only works well on part of the tested data graphs and query graphs, and even a combination of these methods cannot handle all cases well.

Based on this result, we argue that human-designed methods cannot handle all cases well, while the actual matching results, which contain processed structural information of the corresponding data graph, are valuable to help prune the search space in a more general way. The detailed idea is illustrated in Example 1. **Since such pruning uses historical results instead of pre-defined rules, it is orthogonal to existing methods, and thus can work with existing algorithms without conflicts (This is further elaborated in Section 3.3). In general, a system that executes more queries on the same graph can collect more historical results, so it can benefit more from our proposed technique. Two of these application scenarios that shown in Application 1 and Application 2.**

Example 1. *Suppose we are querying all matches of q in d, as Figure 1a shows. A typical search tree of the Ullman's algorithm is shown in Figure 1d. Some branches in the search tree have been pruned by checking the degree of vertices. For example, when the program goes into the branch root $- 3 - 4$, it finds that vertex 4 only has degree 2, while vertex b has degree 3, so vertex b cannot be mapped to 4, and it immediately returns. If we know the matches of some previous queries, we can actually prune more and earlier branches. Here, the matches of a triangle are shown in Figure 1b. According to the matches, we can immediately prune branches root $- 3$ and root $- 4$. This is because vertex a, b, c form a triangle in q, while vertex 3 and 4 cannot form any triangles with any other vertices in d. Similarly, if we know the matches of a rectangle as shown in Figure 1c, we can prune the branch root $- 1$.*



(a) Data graph and query graph. **(b) $\phi_{\triangle}(\{a\})$ of $\triangle(a,b,c)$.** **(c) $\phi_{\square}(\{a\})$ of $\square(a,b,c,d)$.** **(d) A typical search tree.**

Figure 1: The idea of Structure Filtering. All vertices have the same label.

APPLICATION 1. ***Given a graph and an existing subgraph matching algorithm, the user can in advance execute a set of pre-defined common queries, such as triangles and squares. Using our proposed technique, the matching results can be utilized to build a filter database, which can accelerate future queries. The pre-defined queries can be adjusted based on the characteristics of the graph and potential queries, to achieve ideal performance.***

APPLICATION 2. ***Consider a subgraph matching system that continually accepts queries from users. Using our proposed technique, it can build filters with past query results, and then use them to accelerate future queries. With more queries executed, it has a broader choice of filters and is likely to achieve a better performance.***

1.2 Overview and Contributions

Based on the above idea, we propose *StrUcture Filtering Framework* (SUFF), a general and efficient framework that prunes the search space using historical matching results. It can prune the search space not only on a single vertex but also on a set of vertices, which potentially provides more pruning power.

Specifically, this framework consists of two building blocks: a filter database Φ, a subgraph matching algorithm $\mathcal{A}$, and three major steps:

(1) **Filter Selection**. When a new query comes, we select several filters from the database to prune the search space.
(2) **Filtering**. Running $\mathcal{A}$ for subgraph matching, while using selected filters for pruning.
(3) **Filter Construction**: Given the matching result of this query, we construct a set of filters that can help check the presence of vertices, with limited space cost. Those filters are inserted and maintained in the filter database Φ.

In the Filter Selection step, we need to select a few filters to prune the search space. In order to achieve good effectiveness, we propose the *filter utility* model to estimate the pruning power of each filter. It is then proved that maximizing the overall utility is NP-hard, and thus we propose a greedy algorithm that efficiently solves this problem with a lower bound of $1-\frac{1}{\mathbf{e}}$, where $\mathbf{e}$ is the base of the natural logarithm.

In the Filtering step, all selected filters are used to prune the search space of $\mathcal{A}$. Since such pruning happens on the intermediate results of $\mathcal{A}$, it can be done without significantly modifying $\mathcal{A}$.

In the Filter Construction step, given a query q and its matches, we build several filters for different subsets of its vertices. For example, in Figure 1b, we can build a filter for vertex set $\{a\}$ of triangle $\triangle(a,b,c)$, and this filter can be used to check the presence of a single vertex that can be mapped from a. To reduce the space cost of a single filter, we use space-efficient data structures that allow a small false-positive rate, such as Bloom filter [5] or Cuckoo filter [9]. Since there might be many filters stored in the database, we use a filter dropping process to further reduce the storage pressure. Specifically, we propose the concept of *filter domination*, referring to the case that some filters can be replaced with others in any possible situation with a tolerable effectiveness decrease. Using this concept, we propose an algorithm to delete unnecessary filters in polynomial time.

In summary, in this paper, we make the following contributions.

- We propose the novel concept of Structure Filtering, and study some of the basic problems, including its correctness, and when it is applicable.
- We propose a general framework, SUFF, to prune the search space for subgraph matching algorithms which only requires minor modifications to the algorithm.
- We propose the filter utility model to measure the effectiveness of a filter. We prove that maximizing the overall utility is NP-hard, and further propose a greedy algorithm that achieves an approximation ratio of $1-\frac{1}{\mathbf{e}}$.
- We propose the novel concept of *filter domination*, to help reduce the total storage of the filter database. While minimizing the number of stored filters is NP-hard, we propose an efficient greedy algorithm that ensures filter quality across multiple runs.
- We conduct extensive experiments on real-world graphs and several state-of-the-art subgraph matching algorithms. The results show that SUFF achieves up to 15X speedup for existing approaches.

1.3 Paper Organization

The rest of this paper is organized as follows. In Section 2, we introduce the basic concepts in this paper. In Section 3, we investigate the basic properties and conditions of Structure Filtering. In Section 4, we propose the filter utility model that measures filter quality. We then formalize the maximum utility problem for filter selection, and propose a greedy solution. To reduce the space cost, we describe the concept of filter dominating in Section 5, and propose an algorithm to delete unnecessary filters. In Section 6, we conduct an extensive experimental study. At last, we survey the related works in Section 7 and conclude this paper in Section 8.

2 PRELIMINARIES

2.1 Graph and Subgraph

In this work, we focus on the undirected and labeled graph g. $V(g)$ and $E(g)$ denote the vertex set and edge set of g, respectively. Each vertex $v \in V(g)$ is associated with a label $L_g(v)$. An edge connecting two vertices v and u is denoted by (v, u), or equivalently (u, v) as the graph is undirected.

Given a graph g_1, we say g_2 is a *subgraph* of g_1 if $V(g_2) \subseteq V(g_1)$ and $E(g_2) \subseteq E(g_1)$. Specifically, g_2 is the subgraph of g_1 *induced by* vertex set V' if (1) $V(g_2) = V' \subseteq V(g_1)$, and (2) $E(g_2) = E(g_1) \cap (V' \times V')$. In this case we also call g_2 an *induced subgraph* of g_1, and denote it by $g_1[V']$. Apparently any graph g is the subgraph of itself induced by $V(g)$.

2.2 Subgraph Matching

By introducing the concept of graph isomorphism, we can then formally define the subgraph matching problem.

Definition 1 (Graph Isomorphism [36]). *Given two graphs g_1 and g_2, an isomorphism from g_1 to g_2 is a bijection $f : V(g_1) \mapsto V(g_2)$ such that (1) $L_{g_1}(v) = L_{g_2}(f(v))$, and (2) $(u, v) \in E(g_1)$ if and only if $(f(u), f(v)) \in E(g_2)$. If there is an isomorphism from g_1 to g_2, then we say g_1 is isomorphic to g_2.*

Definition 2 (Subgraph Matching). *Given two connected graphs g_1 and g_2, subgraph matching requires to find all subgraphs of g_1, which are isomorphic to g_2. Here g_1 is also called the data graph, and g_2 is also called the query graph.*

For the ease of description, in the rest of this paper, we use d to denote the data graph, and q to denote the query graph. Each isomorphism f from q to a subgraph of d is called a *match*.

2.3 Search Tree and Partial Result

As discussed in Section 1, the basic approach of subgraph matching is a brute-force search. Thus, almost all existing solutions traverse the search space following a search tree (or several search trees). During this process, each internal node in the search tree(s) corresponds to a mapping assignment from a subset of $V(q)$ to $V(d)$. Such an assignment is called a *partial match*.

Definition 3 (Partial Match). *A partial match of query q in graph d is a match $f_p[V] : V' \mapsto V(d)$, where $V' \subseteq V(q)$.*

For example, in Figure 1d, the branch $root - 1 - 2$ corresponds to the partial match $\{a \mapsto 1, b \mapsto 2\}$.

2.4 Filter

A filter ϕ is a (space-efficient) representation of a set $\mathcal{S}$. Given any element $e \in \mathcal{S}$, ϕ *accepts* (returns positive) e for sure. Given any element $e \notin \mathcal{S}$, ϕ accepts e with probability p, and *rejects* (returns negative) e with probability $1 - p$. Here p is known as the false-positive rate. Filters are widely used when the set $\mathcal{S}$ is too large to store, and a certain false-positive rate is acceptable.

In this work, we use Bloom Filter [5] as the underlying implementation. The reason is that, its overhead is low, and the intersection of multiple Bloom Filters can be easily obtained to reduce some computation cost. To control the false-positive rate of each filter not to be too high, we use a pre-defined threshold p_{max}, and drop the filters with $p > p_{max}$. In the rest of this paper, we do not distinguish between a filter and a set when p is sufficiently small, and only discuss p when necessary.

3 THE FILTERING FRAMEWORK

In this section, we first analyze the relationship between matches of two query graphs, and then describe how SUFF prunes the search space. After that, we briefly discuss how to utilize filters to improve the space and time efficiency for existence check, and further give the framework overview.

3.1 The Principle of Structure Filtering

The basic idea of structure filtering is to use the matches of a small structure (query) to prune out search branches of a large structure (query). Given two query graphs q and q', as well as their match sets $M(q, d)$ and $M(q', d)$, if q' is a subgraph of q, then we have the following lemma.

Lemma 1. *For any match $f \in M(q, d)$, there exists a match $h \in M(q', d)$, such that $\forall v \in V(q') : f(v) = h(v)$.*

Proof. Since q' is a subgraph of q, we know that $V(q') \subseteq V(q)$, and $E(q') \subseteq E(q)$. Let $h = v \mapsto f(v), \forall v \in V(q')$, it is trivial that $\forall (v, u) \in E(q') : (h(v), f'(u)) \in E(d)$, so $h \in M(q', d)$. □

By Lemma 1, if there exists no such h, we can conclude that f is not a valid match immediately. This gives the basis for using the match set of q' to prune the search space for q. However, when f is completely constructed during the search, most existing algorithms already verified its validity, so there is no need to perform the structure filtering. Similarly, when h is completely constructed as part of f during the search, its validity is also verified by the algorithm itself, so it must be a valid match of q'. Therefore, in order to accelerate the algorithm, we need to prune a partial match before the original program can verify the validity of f or h. The following lemma suggests that we can extend structure filtering to any partial match of q, which is more useful.

Lemma 2. *Given a match $f \in M(q, d)$, let $f_p[V]$ be a partial match of f, where $V \subset V(q') \cap V(q)$, then there exists a match $h \in M(q', d)$, such that $\forall v \in V : f_p(v) = h(v)$.*

Proof. It is trivial since Lemma 1 ensures the existence of h. □

Note that in Lemma 2, V is a subset of $V(q')$. This means during the matching process, we can use the existence of partial match h_p to prune partial match f_p. Whenever a partial match f_p is generated, if we cannot find such a corresponding h in $M(q', d)$, we can stop in this branch and return. Since $V \subset V(q') \cap V(q)$, we are pruning branches before the algorithm can verify the validity of h or f. This is essentially the power of structure filtering.

Remark 1. *Even if a partial match f_p passes our checking, it is still possible that this branch doesn't contain any valid match of q (a false-positive case).*

In SUFF, when $M(q, d)$ is produced, filters are constructed for different subsets of $V(q)$, which correspond to different partial matches (We will discuss the details in Section 3.4). They are then selected and used to accelerate the matching of future queries.

3.2 Utilizing Filters

A simple way to record partial matches for existence check is to store them in a plain set. However, this introduces high storage pressure. **For example, storing 10^5 3-vertex partial matches takes more than 1MB of disk space, and there could be thousands of such match set to store.**

In SUFF, we use filters to achieve both space and time efficiency. Specifically, given a match set $M(q, d)$, we build a filter $\phi_q(V)$ for each $V \subset V(q)$. Each filter $\phi_q(V)$ stores all the partial matches in $\{h_p[V], \forall h \in M(q, d)\}$. When we need to check the existence of an h given $f_p[V]$, we can directly check whether $f_p[V]$ is in $\phi_q(V)$. When the context is clear, we also use ϕ to refer to a filter.

By storing those matches in filters, we can save much disk space. To store a match set containing 10^5 matches, if we use a Bloom Filter with a false-positive rate of 0.1, we only need about 60KB of storage.

Another benefit of using filters is the efficient insertion and checking operations. Also take Bloom Filter as an example, to ensure a false-positive rate of 0.1, each insertion/checking operation only involves 3 hash functions under a common setting, and can be treated as $O(1)$ operation. This ensures that SUFF does not introduce too much overhead compared to the original algorithm.

To guarantee the space cost and effectiveness of each filter, we use two user-specified parameters m and p_{max}. Then each created filter occupies m bits of storage, and we only use filters with a false-positive rate less than or equal to p_{max}

It should be noted that, filters do not yield false-negative results. Therefore, if the filter rejects a partial match, we can safely drop it. This is consistent with our structure pruning technique.

3.3 Framework Overview

With the above basic building blocks, we now describe how SUFF works. In general, SUFF works with a subgraph matching algorithm $\mathcal{A}$, and maintains a filter database Φ, which is used and updated when processing a new query q through the following steps:

(1) **Filter Selection**. Pick several filters from Φ.
(2) **Filtering**. When running $\mathcal{A}$ for subgraph matching, we use these picked filters as early as possible for structure filtering.
(3) **Filter Building**. We use the subgraph matching result to build filter $\phi_q(V)$ for each $V \subset V(q)$. these filters are added to the database Φ.

In the first step, we need to make sure that, each picked filter $\phi_{q'}(V)$ is usable for q, which means

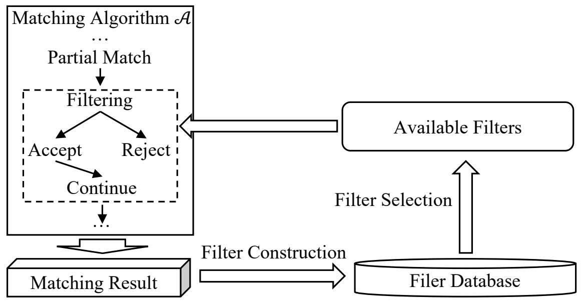


Figure 2: The overview of SUFF.

(1) q' is a subgraph of q; and
(2) this filter can be used before a full match of q is generated.

Therefore, it is important to associate a match $h : V(q') \mapsto V(q)$ with each filter for verifying these two conditions. If there are multiple matches of q' in q, we create multiple filters for each of them. In order to achieve better pruning performance, we also want to pick the best set of filters, which is discussed in Section 4.

Algorithm 1: A Typical Modified Matching Algorithm

```
Input  : Data graph $d$, query graph $q$, filter database $\Phi$.
Output : All matches of $q$ in $d$.
1  $\mathcal{D}$ ← generate auxiliary data structure;
2  $F$ ← selected filters from $\Phi$;
3  Enumerate($d$, $q$, $\mathcal{D}$, {}, 1);
4  Procedure Enumerate($d$, $q$, $\mathcal{D}$, $f$, $i$):
5      if $i = |V(q)| + 1$ then
6          Output $f$;
7          return;
8      if any filter in $F$ rejects $f$ then return;
9      $u$ ← the query vertex to be matched in this level;
10     $C$ ← generate candidates for $u$;
11     foreach $v \in C$ do
12         if $v \notin f$ then
13             Add $\{u \mapsto v\}$ to $f$;
14             Enumerate($d$, $q$, $\mathcal{D}$, $f$, $i$+1);
15             Remove $\{u \mapsto v\}$ from $f$;
```

While selecting the filter set, we find when each filter can be used in the matching process. This can be done by looking at the search plan of $\mathcal{A}$, and finding the smallest depth where a filter can be applied. We denote this depth as the filter's *filtering level*. Then we run $\mathcal{A}$ for subgraph matching, and apply each filter at its filtering level. As most existing approaches find matches in a depth-first-search manner, the filtering can be plugged into them with minor modifications. We show a representative example in algorithm 1, where the shadowed areas are the modification made by SUFF. This algorithm uses a recursive subroutine "enumerate" to find matches of q. Each recursive call extends the current match f by adding a mapping $\{u \mapsto v\}$. When it reaches the maximum level ($|V(q)|+1$), f is a complete valid match of q. To let SUFF work, we only need to select a set of filters from the database (line 2), and check the (partial) match f at each level (line 9). If f is rejected, the recursive call immediately returns.

Finally, given the match set $M(q, d)$, we build filters for subsets of $V(q)$, and store them into Φ (Section 3.4). Those filters can be used when there are new coming queries. To save the computation resource as well as the disk space, we use a filtering dropping step to eliminate some redundant filters, so that the number of filters in Φ does not grow too quickly, which is discussed it in Section 5.

3.4 Filter Construction and Storage

A filter $\phi_q(V)$ is essentially a bloom filter storing all partial matches $f_p[V]$ of q, so one can build a filter for each subset $V \subset V(q)$. However, this is space inefficient, since the number of filters is exponential to $|V(q)|$. In this work, we make three observations that help reduce the number of filters.

OB$_1$ The number of nodes in the search tree is usually larger when the level goes deeper, so filters would have larger computation overhead;

OB$_2$ The matching orders of q and q' tend to be similar if q and q' are similar.

OB$_3$ Filters built with more vertices ($|V|$) tend to have better filtering power, since the stored partial matches are more specific.

Based on the above observations, we can avoid constructing filters that are expected to be applied on deep levels, and should make the best use of all vertices. Specifically, our strategy is as follows.

- **Let a query have m vertices, and $v_1, v_2, \cdots, v_m$ be the matching order decided by an existing algorithm;**
- **For any number $a \in [1, m]$, we can generate two kinds of filters with the sequence $v_1, v_2, \cdots, v_a$. (1) We build one filter for each prefix of the sequence, i.e., $\{v_1\}$, $\{v_1, v_2\}, \cdots, \{v_1, v_2, \cdots, v_a\}$; (2) We build one filter for each single vertex in $v_2, v_3, \cdots, v_a$;**
- **In total, we build $2 \cdot a - 1$ filters for a query, and the user can pick appropriate a according to the space limitation.**

In this way, we expect that filters can be applied at the earliest possible levels when the query q is similar to the filter pattern, and the pruning power of multi-vertex filters can be best utilized at deep levels.

To store all filters, we design a hybrid file storage. The configurations of all filters, such as the pattern shapes, vertex sets, and estimated false-positive ratios, are stored in a single file. This information takes only a few bytes for a filter, so the program can read them all from a file quickly. Meanwhile, we store the underlying bit array of each filer in a separate file. These files would be read only when the corresponding filters are selected to use, so we can avoid unnecessary I/Os.

4 FILTER SELECTION

Given a specific query q, different filters may have different pruning power. The computation cost we spent on each filter also varies due to the size of its vertex set. Apparently, using more filters potentially gives more pruning power, but the overhead also grows up, and may in turn slow down the algorithm. Thus, in order to obtain a better performance, it is important to carefully select filters. In SUFF, we strike a balance between benefits and overheads by a comprehensive filter selection process.

Specifically, we build a comprehensive filter utility model to measure the pruning power of each filter, which considers both the query shape and the execution plan of $\mathcal{A}$. Then we try to maximize the overall utility by picking a limited number of filters.

4.1 The Filter Utility Model

As discussed above, filters can prune the search space before the original algorithm can validate all edges. Thus, the less validated edges, the more pruning power (or utility). In SUFF, we build a simple yet effective utility model based on this idea.

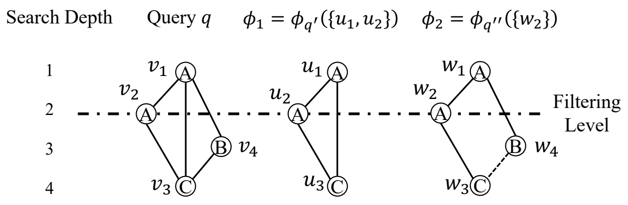


Figure 3: An illustration of filter utility breakdown.

Example 2. ***In Figure 3, we show a query q to be matched in the order v_1, v_2, v_4, v_3. Filter ϕ_1 is built on vertex set $\{u_1, u_2\}$ from the matches of q' (a triangle), and ϕ_2 is built on $\{w_2\}$ from the matches of q'' (a square). Both filters can be applied at level 2. At this level, mappings of vertices v_1, v_2 are fixed, so the edge (v_1, v_2) is validated. Other edges' existence is unknown. If we apply ϕ_1, it can verify whether a triangle containing $f(v_1), f(v_2)$ exists, which corresponds to a valid mapping of v_3. If no such triangle exists, the program can directly return without going to level 3. In other words, it helps validate edges (v_2, v_3) and (v_1, v_3). Similarly, applying ϕ_2 helps validate edges (v_2, v_3), (v_1, v_4), and (v_3, v_4). Applying both filters can help validate all rest edges in q.***

Example 2 briefly shows how we break down filter utility into edges. Doing so enables us to quantify not only the utility of a single filter, but also the utility of combined filters. **In addition, our model considers the *neighborhood label frequency filtering* technique which is widely used in existing solutions. With this technique, the algorithm ensures that each candidate vertex satisfies the neighbor label frequency in the query graph. In Figure 3, when v_1, v_2 are mapped, the algorithm ensures that $f(v_1)$ has a neighbor with type B and a neighbor with type C, and $f(v_2)$ has a neighbor with type C. In this case, all edges except for (v_3, v_4) can be treated as validated, so ϕ_1 is considered less helpful.**

Our utility model is formally defined as follows. Given a matching order and a filtering level l, $V(q, l)$ denote the vertex set that are mapped by the algorithm (which is $\{v_1, v_2\}$ in the example), and $\overline{V}(q, l) = V(q) \setminus V(q, l)$ is the unmapped vertex set. Then for a filter

ϕ of pattern q' and a corresponding match h, its utility on edge $e = (u, v)$ is

$$S(\phi, l, e) = \mathbb{1}_{h(u) \in \overline{V}(q,l) \wedge h(v) \in \overline{V}(q,l)}. \quad (1)$$

Then for a set of filters F_l at this level, the overall utility is

$$S(F_l) = \sum_{e \in E(q)} \max_{\phi \in F_l} S(\phi, l, e). \quad (2)$$

Under this model, ϕ_1 has a utility score 0, because all edges in its triangle pattern connect to at least one mapped vertex at its filtering level. ϕ_2 has utility score 1, because only one edge (w_3, w_4) connects to no mapped vertex.

This function is built for three reasons: (1) it considers the situation that multiple filters may help on the same edge, and avoids double counting the utility; (2) it is easy to compute, which is helpful when the filter database is large; and (3) this function naturally eliminates filters with filtering level= $|V(q)|$ by assigning utility score 0 to them, which are shown to be useless in Section 3.1.

4.2 The Maximum Utility Problem

As discussed above, our goal is to pick a set of filters such that the overall performance is maximized. Since filters at different levels are not directly related, we aim at maximizing the total utility score $S(F_l)$ for each level l independently, which is so-called the maximum utility problem.

Definition 4 (Maximum Utility Problem). *Given query q, a filter database Φ, a matching order, and an integer k, the maximum utility problem is to select a set of filters F_l from Φ for each level l, s.t.*

(1) $|F_l| \leq k$, and
(2) the total utility score $S(F_l)$ for each F_l is maximized.

Unfortunately, Theorem 1 shows that the maximum utility problem is NP-hard by reducing from a well-known NP-hard problem, set cover [18].

Theorem 1. *The maximum utility problem is NP-hard.*

Proof. Consider the decision version of the maximum utility (DMU) problem, which is to decide whether we can select k filters from Φ, so that $S(F_l) \geq s$. We show that it can be reduced from the decision version of the set cover (DSC) problem.

The DSC problem can be described as follows. Given a universe of elements $U = \{e_1, e_2, \ldots, e_n\}$, and a collection $C = \{c_1, c_2, \ldots, c_m\}$ of m sets, where $\bigcup_{c_i \in C} c_i = U$, and an integer k, it is to decide whether there exists $C' \subseteq C$, s.t. $|C'| = k$ and $\bigcup_{c \in C'} c = U$.

Given an instance of DSC, we can build an instance of the DMU in polynomial time. First, we create a graph q, where $V(q) = \{v_i | 0 \leq i \leq n\}$ and $E(q) = \{(v_0, v_i) | 1 \leq i \leq n\}$. That is, a star-shaped graph with v_0 as the center. Each vertex v_i has a unique label t_i. This can be done in $O(n)$ time. Then for each set $c_j \in C$, we put a filter $\phi_{q_j}(\{v_0\})$ into Φ. Its corresponding pattern is also a star-shaped graph with u_0 as the center, where $V(q_j) = \{u_i | \forall e_i \in c_j\} \cup \{u_0\}$ and $E(q_j) = \{(u_0, u_i) | \forall u_i \in V(q_j)\}$. Each vertex u_i also has label t_i. This can be done in $O(m \cdot n)$ time, as there are m sets, each with size $\leq n$. Finally, we set v_0 as the first vertex to match, and $s = n$, and we have all components of an instance of the DMU problem.

Next we show how the solutions of these two problems are related. It is easy to verify that ϕ_{q_j} corresponds to set c_j in the DSC problem. If ϕ_{q_j} is selected, the utility score for edge $(u_0, u_i), \forall e_i \in c_j$ is 1 in Equation 2. Thus selecting ϕ_{q_j} is equivalent to selecting c_j in the DSC problem. If the answer to the DMU problem is yes, the set $C' = \{c_j | \forall \phi_{q_j} \in F_0\}$ can cover all elements in U, and the answer to the DSC problem is also yes. In reverse, if the answer to the DSC problem is yes, we can set $F_0 = \{\phi_{q_j} | c_j \in C'\}$, then its utility score is n, so the answer to the DMU problem is also yes. Note that each selected filter corresponds to a set $c_i \in C$, so a solution of the maximum utility problem of size k which has total utility score s corresponds to a subset of C in the set cover problem, who has size k and covers s elements in total.

In summary, the DMU problem can be reduced from the DSC problem in polynomial time, so it is NP-complete, and the maximum utility problem is therefore NP-hard. □

Since it is unlikely that we can solve the maximum utility problem in polynomial time, we use a greedy algorithm in SUFF to balance the running time and result quality. In this algorithm, instead of finding the global optimal, we iteratively pick filters while maximizing the utility score at each step. Algorithm 2 summarizes the greedy algorithm. The whole algorithm runs for k iterations in total (line 2). At each iteration, we select one filter in Φ to add into the current solution, such that the total utility score of the current solution is maximized (lines 3 to 12).

Algorithm 2: FilterSelectGreedy

```
Input  : $\Phi$, $q$, the matching order, $k$, $l$.
Output : $F_l$.
1  $F_l \leftarrow \emptyset$;
2  while $|F_l| < k$ do
3      $currentBestScore \leftarrow S(F_l)$;
4      $selectedFilter \leftarrow null$;
5      foreach $\phi^{(i)} \in \Phi$ do
6          if $\phi^{(i)}$ is not applicable at this level then
7              continue;
8          $F'_l \leftarrow F_l \cup \{\phi^{(i)}\}$;
9          if $currentBestScore \leq S(F'_l)$ then
10             $currentBestScore \leftarrow S(F'_l)$;
11             $selectedFilter \leftarrow \phi^{(i)}$;
12     if $selectedFilter = null$ then
13         break;
14     $F_l \leftarrow F_l \cup \{selectedFilter\}$;
15 return $F_l$;
```

As stated above, the algorithm runs for at most k iterations. In each iteration we need to try adding every filter in Φ and calculate the utility score $S(F'_l)$. Calculating $S(\phi, l, e)$ takes $O(1)$ time, so it takes $O(k \cdot |E(q)|)$ time to calculate $S(F_l)$. Therefore, the time complexity of the greedy algorithm is $O(k^2 \cdot |\Phi| \cdot |E(q)| \cdot \tau)$, where τ is the time for subgraph matching between the query graph q and a filter pattern. We note here that, since the query graphs are supposed to be small (i.e., < 100 nodes) in all existing solutions, τ

is small. Therefore, the running time of algorithm 2 is negligible compared with the query processing time.

Another advantage of this greedy algorithm is that, it also guarantees the quality of the result. In fact, Theorem 2 shows that, its result is at least $1-\frac{1}{\mathrm{e}}$ of the optimal result, **where e (distinguished from an edge e) is the base of the natural logarithm.**

Theorem 2. *Algorithm 2 has $1-\frac{1}{\mathrm{e}}$ approximation ratio.*

Proof. apparently $S(F_l)$ is a monotone function. We further show that it is also submodular.

Let $T(F_l, e) = \max_{\phi \in F_l} S(\phi, l, e)$, $F \subseteq F'$, and $X = F' \setminus F$, and Y be an arbitrary set. If $S(F' \cup Y, e) = S(F \cup Y, e) = 0$, we have

$$T(F' \cup Y, e) - T(F', e) = T(F \cup Y, e) - T(F, e) = 0.$$

If $S(F' \cup Y, e) = S(F \cup Y, e) = 1$, then

$$\begin{aligned} T(F' \cup Y, e) - T(F', e) &= T(F \cup Y, e) - T(F', e) \\ &\leq T(F \cup Y, e) - T(F, e). \end{aligned}$$

If $S(F' \cup Y, e) = 1$ and $S(F \cup Y, e) = 0$, we know that $S(F', e) = 1$ and $S(F, e) = 0$, and thus

$$T(F' \cup Y, e) - T(F', e) = T(F \cup Y, e) - T(F, e) = 0.$$

All cases indicate that $T(F' \cup Y, e) - T(F', e) \leq T(F \cup Y, e) - T(F, e)$, which means $T(F_l, e)$ is a submodular function. Hence $S(F_l) = \sum_{e \in E(q)} T(F_l, e)$ is a sum of independent submodular functions, which is also submodular.

Since $S(F_l)$ is a monotone submodular function, and algorithm 2 is greedily maximizing the function value iteratively, the result has an approximation bound of $1-\frac{1}{\mathrm{e}}$ according to [21]. □

5 FILTER REMOVAL

As more queries are processed, more filters are built and added into Φ, so the size of Φ keeps increasing. This not only consumes much disk space, but also harms the efficiency of filter selection. However, some filters are redundant and can be replaced by other filters given any query. Removing these filters in advance would help keep the whole framework efficient. In this section, we discuss how SUFF identifies the redundant filters, and how to remove them from Φ while keeping a high pruning effectiveness.

5.1 Filter Domination

If we want to remove a filter from the database, the risk is that the filtering power of SUFF for some queries is weakened due to the lack of this filter. Thus, it is important to ensure that the side effect of removing filters is minimal. Following this principle, we investigate the domination between filters.

Definition 5 (Filter Domination). *Given two filters $\phi_o(V_1)$ and $\phi_r(V_2)$, we say $\phi_r(V_2)$ is **dominated by** $\phi_o(V_1)$ if and only if for any query q, the following conditions always hold:*

- DC_1 *whenever $\phi_r(V_2)$ can be applied for filtering, $\phi_o(V_1)$ can be applied at the same filtering level; and*
- DC_2 *the false-positive rate of replacing $\phi_r(V_2)$ with $\phi_o(V_1)$ remains $\leq p_{max}$.*

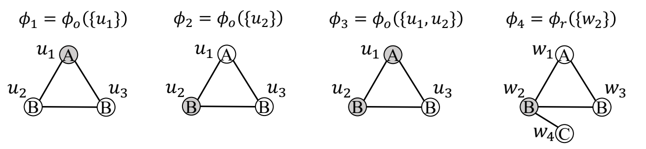

$\phi_1 = \phi_o(\{u_1\})$
$\phi_2 = \phi_o(\{u_2\})$
$\phi_3 = \phi_o(\{u_1, u_2\})$
$\phi_4 = \phi_r(\{w_2\})$
u_1 A
u_2 B
u_3 B
w_1 A
w_2 B
w_3 B
w_4 C

Figure 4: An example of filter domination.

Example 3. *In Figure 4 there are four filters. Filters ϕ_1, ϕ_2, ϕ_3 have the same pattern graph o (the triangle), while ϕ_4 has a pattern r that is a super graph of the triangle. Apparently, if ϕ_4 can be applied for a query, the other three can also be applied. In addition, ϕ_2 can be applied on the same level as ϕ_4 does, because u_2 corresponds to w_2 in one of the matches from o to r. ϕ_1 and ϕ_3 will have a different filtering level if w_1 is after w_2 in the matching order, so they cannot replace ϕ_4 in some cases. Therefore, if u_2 and w_2 are mapped to roughly the same vertex set in ϕ_2 and ϕ_4 respectively, ϕ_2 can always replace ϕ_4 with little harm to the filter effectiveness. Another possible situation is that, when we have both ϕ_1 and ϕ_2 in the database, it is always applicable to use one to replace ϕ_3 at the same level. If the mapped sets of u_1 and u_2 happen to be close across these three filters, such replacement is also almost harmless.*

In other words, a filter is dominated if it can always be replaced by another one without harming the effectiveness. In order to detect the domination relationship between filters, we further investigate the detailed rules that can be used to detect filter domination. Lemma 3 shows that condition DC_1 holds if and only if (1) o is a subgraph of r, and (2) V_1 is a subset of V_2.

Lemma 3. *DC_1 holds for any query q if and only if o is a subgraph of r and $V_1 \subseteq V_2$.*

Proof. **(If)** Applying $\phi_r(V_2)$ on $f_p[V]$ requires that r is a subgraph of q and $V_2 \subseteq V$, thus o is also a subgraph of q and $V_1 \subseteq V$, which means $\phi_o(V_1)$ can also be applied on $f_p[V]$.

(Only if) Let $q = r$ and $V_2 = V$, if $\phi_o(V_1)$ can be applied on $f_p[V]$, then o is a subgraph of r and $V_1 \subseteq V_2$. □

Using Lemma 3, we can quickly find all possible filters in Φ that might replace a specific filter. Then we need to check if condition DC_2 also holds for any of them. That is, we need to discuss about the false-positive rate of applying $\phi_o(V_1)$ instead of $\phi_r(V_2)$. Our discussion splits DC_1 into two cases: $V_1 = V_2$ and $V_1 \subset V_2$.

($V_1 = V_2$). As discussed in Section 3.1, any match f of r corresponds to a match f' of o. Thus, for each partial match $f_p[V_2]$ we store in $\phi_r(V_2)$, there is a same partial match $f'_p[V_1]$ in $\phi_o(V_1)$. This means $\phi_o(V_1)$ is a super set of $\phi_r(V_2)$. Let N be the total number of items, p_o and p_r be the false-positive rate, N_o and N_r be the numbers of items for $\phi_o(V_1)$ and $\phi_r(V_2)$, respectively. Here p_o, p_r, N_o, N_r can be recorded when building the filter, or estimated using the bit table of the filter. If we use $\phi_o(V_1)$ to replace $\phi_r(V_2)$, the false-positive items come from two parts. One part is the items exist in $\phi_o(V_1)$ but not in $\phi_r(V_2)$, which contains $N_o - N_r$ items in total. Another part is the items accepted falsely by $\phi_o(V_1)$, which contains $(N - N_o) \cdot p_o$ by expectation. Therefore, the total false-positive rate is

$$p' = \frac{(N - N_o) \cdot p_o + N_o - N_r}{N - N_r} = p_o + \frac{(N_o - N_r)(1 - p_o)}{N - N_r}.$$

Note that we do not know how large N is, but $\phi_r(V_2)$ must be able to filter out certain portion of N in order to be effective (otherwise applying $\phi_r(V_2)$ or $\phi_o(V_1)$ does not make much difference). Hence it is reasonable to assume that $N \geq \alpha \cdot N_r$, where α is a desired pruning ratio that could be adjusted. Then we have an upper bound of p':

$$upper(p') = p_o + \frac{(N_o - N_r)(1 - p_o)}{(\alpha - 1) \cdot N_r}. \quad (3)$$

If this upper bound is smaller than p_{max}, $\phi_o(V_1)$ dominates $\phi_r(V_2)$. A larger α makes the estimated false-positive ratio p' smaller, so it is more likely to satisfy DC$_2$ and to remove filters and the size of Φ is kept smaller. However, this also lower down the expected pruning power of remaining filters.

($V_1 \subset V_2$). In this case, although we can apply two filters at the same level, they checks different sets of items (vertex sets). Thus, the analysis above cannot be directly used since $\phi_o(V_1)$ is no longer a super set of $\phi_r(V_2)$. However, if $o = r$, mappings in $\phi_o(V_1)$ are partial mappings of those in $\phi_r(V_2)$. If the size of these two sets are close, it means using only part of each mapping in $\phi_r(V_2)$ can still distinguish the whole set with high accuracy, and replacing $\phi_r(V_2)$ with $\phi_o(V_1)$ is doable. In this situation the above analysis still applies.

In summary, $\phi_o(V_1)$ dominates $\phi_r(V_2)$ if and only if

(1) $upper(p') \leq p_{max}$, and
(2) $V_1 \subseteq V_2$, and
(3) either $o = r$, or $V_1 = V_2$.

5.2 The Filter Removal Problem

The process for detecting and removing dominated filters can be run periodically or whenever new filters are inserted. Either way will not influence the matching process. Note that, the domination is not a transitive relation, i.e., if A dominates B and B dominates C, A may not dominate C. If that is the case, we can either remove B or C, but cannot remove them both. This makes it challenging to minimize the number of filters in Φ, which is proved to be NP-hard.

Theorem 3. *Given a set Φ of filters and their domination relationships, to find a minimum subset S of Φ s.t. every filter in Φ either belongs to S or is dominated by a filter in S, is NP-hard.*

Proof. We prove this by reducing from the set cover problem. Given a universe of elements $U = \{e_1, e_2, \dots, e_n\}$, and a collection $C = \{c_1, c_2, \dots, c_m\}$ of m sets, where $\bigcup_{c_i \in C} c_i = U$, it is to find a set $C' \subseteq C$, s.t. $|C'|$ is minimized and $\bigcup_{c \in C'} c = U$.

Given an instance of the set cover problem, we create a filter set $\Phi = \{\phi^i | 0 \leq i \leq n + m\}$ containing $n + m + 1$ filters. Filter $\phi^i, \forall 1 \leq i \leq m$ corresponds to c_i, and filter $\phi^{m+j}, \forall 1 \leq i \leq n$ corresponds to e_j. The domination relationship is decided as follows. For each c_i, ϕ^0 dominates ϕ^i, and for each element $e_j \in c_i$, ϕ^i dominates ϕ^{m+j}. Now we get an instance of the filter set minimization problem in $O(n \cdot m)$ time.

Suppose S is a solution of the filter set minimization problem, we show that there is a corresponding solution to the set cover problem of size $|S| - 1$. First, ϕ^0 must be in S since it is not dominated by other filters. Then we construct C' as follows. For each $\phi^i \in S \setminus \{\phi^0\}$, if $1 \leq i \leq m$, we add c_i into C', otherwise we find a ϕ^j that dominates ϕ^i, and add c_j into C'. Note that there will not be duplicated c_js that we add into C', otherwise the set $|S|$ is not minimal. So $|C'| = |S| - 1$. Since every filter $\phi^m + i, 1 \leq i \leq n$ is either in S or is dominated by a filter in S, its corresponding element e_i is covered by a set in C'. Thus, C' is a valid solution of the vertex cover. □

Due to the hardness of this problem, we devise a greedy algorithm (Algorithm 3) to remove redundant filters to improve efficiency. This algorithm also considers the consistency across multiple independent runs. Suppose A dominates B, B dominates C, but A does not dominate C. If C is removed in former runs, it is important to remember not to remove B in later runs. To achieve this, we record for each filter a *dominating set*, containing information of filters that we use it to replace. The information is used to calculate Equation 3. And we extend the definition of domination by one condition that, a filter must be able to replace all filters in another filter's dominating set.

Algorithm 3: FilterRemoval

```
Input  : The filter database $\Phi$.
Output : The minimized database.
1 $D \leftarrow$ build the dominating graph between filters (A DAG);
2 foreach $\phi^i \in D$ in bottom-up order do
3     if $\phi^i$ is dominated by $\phi^j$ then
4         $\Phi \leftarrow \Phi / \{\phi^i\}$;
5         Add $\phi^i$ into $\phi^j$'s dominating set;
6         Update the domination relationship related to $\phi^j$;
7 return $\Phi$;
```

Apparently there could not be cyclic domination relationships according to our analysis, so the algorithm build a Directed Acyclic Graph (DAG) to guide the order of removing filters. Filters in lower level are dominated by other filters, and thus are less general to be applied in future queries. Thus, the algorithm tries to remove these filters first, and then those in upper levels. Checking the domination requires at most $O(|\Phi|)$ time when the dominating set is large, so the time complexity of this algorithm is $O(|\Phi|^2 \cdot \tau)$, where τ is the time for matching two filter patterns (as we discussed before, τ is small since the query graphs are usually small). As this process can run in offline, it will not affect the efficiency of query processing.

6 EXPERIMENTS

In this section we conduct extensive experiments to verify the effectiveness of SUFF across different data sets and state-of-the-art solutions.

6.1 Experimental Setup

Data Sets. We use eight real-world representative graph data sets that are commonly used in the literature [34] for testing synthetic queries, and a knowledge graph DBpedia [2] for testing real-world queries. The statistics is summarized in

Table 1: Statistics of Real-world Data Sets.

Data Set	$\|V\|$	$\|E\|$	d	$\|L\|$
Yeast (ye)	3,112	12,519	8.0	71
Human (hu)	4,674	86,282	36.9	44
HPRD (hp)	9,460	34,998	7.4	307
WordNet (wn)	76,853	120,339	3.1	5
DBLP (db)	317,080	1,049,866	6.6	15
eu2005 (eu)	862,664	16,138,468	37.4	40
Youtube (yo)	1,134,890	2,987,624	5.3	25
US Patents (us)	3,774,768	16,518,947	8.8	20
DBpedia	62,508,248	300,379,692	9.6	483734

Table 2: Space cost of SUFF.

Graph	Graph Size	Filter Database Size
Yeast	0.17MB	22.82MB
Human	0.96MB	22.82MB
HPRD	0.49MB	22.82MB
WordNet	2.5MB	3.91MB
DBLP	21MB	91.3MB
eu2005	277MB	91.3MB
Youtube	63MB	91.3MB
US Patents	360MB	91.3MB
DBpedia	6144MB	58.48MB

Table 1. In the table, $|V|$ and $|E|$ are the number of vertices and edges, respectively. d is the average degree and $|L|$ is the number of labels. For the first eight data graphs, we generate query sets by randomly selecting subgraphs from it, which is consistent with previous studies [3, 11, 19, 34]. Specifically, we generate both sparse queries (average degree < 3) and dense queries (average degree ≥ 3). **The query size $|V(q)|$ varies in $\{8, 16, 24, 32\}$.** In total, there are 800 queries for each data set. **For DBpedia, we extract graph patterns from real-world SPARQL logs [33]. After removing invalid queries (e.g., unconnected queries), there are in total 3731 queries in our experiments.**

Implementation and Parameter Settings. All algorithms are implemented in C++ and are compiled with g++ 11. Experiments are run on a single machine with an Intel Xeon X5650 CPU and 100GB memory. **For filters, to save space, we only create filters with no more than 3 vertices ($a = 3$) using our strategy.** To balance the space cost and the computation cost, we set the number of hash functions of each bloom filter to be 3, and the expected false-positive ratio to be 0.01. **According to the graph size, we set the bit array size to be 1KB for Yeast, Human, HPRD, 2KB for WordNet, 8KB for DBpedia, and 4KB for other graphs.** For filter removal, we set $\alpha = 0.3$.

Evaluated Methods. We evaluate our proposed filtering framework together with six representative subgraph matching methods QuickSI (QSI) [30], GraphQL (QGL) [13], VF2++ [15], CECI [3], CFL [4], DP-iso (DP) [11]. For each algorithm X, we plugin the SUFF framework, and mark it as X/S-i, where $i \in [1, 3]$ is the number of filters used for each level. For each graph, we build an initial filter database using 4 basic queries: the triangle-shaped query (three-vertex cycle) and the 4-vertex query (four-vertex cycle, diamond graph, and 4-clique). specifically, we extract the top-10 frequent labels for each graph, and enumerate all shapes with these labels. Since these queries have a few vertices, building the databases only take several minutes. In total, there are at most 4675 pattern graphs for each data set. In addition, DP-iso uses dynamic matching order, so we utilize matching orders provided by other methods, and calculate the average filter utility in the filter selection step. And filters are applied adaptively for each branch based on the current matching order.

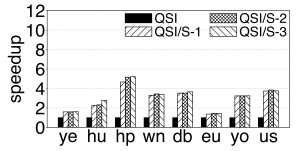
Metrics. For synthetic queries, we count the query processing time (wall-clock time) of each algorithm, and report the average *speedup* of the modified algorithm vs. the original algorithm in 100 independent runs. Here *speedup* is defined as $speedup(A', q) = \frac{A(q)}{A'(q)}$, where $A(q)$ is the processing time of the original algorithm and $A'(q)$ is the time of the modified algorithm. That is, if A' costs half of the time as A does, the speedup is 2. We choose this metric instead of the average run time because the processing time of different queries can differ in orders of magnitude, and the average run time hides the acceleration effect on short-time queries. As some queries take a long time to finish, we set a time limit of 1000 seconds to let the experiments finish in a reasonable time. To make the results more informative, if an original method fails to finish a query within the time limit, we exclude all results of this query and this method with/without SUFF. We also measure the effectiveness of SUFF by counting the number of failing branches (branches that do not yield any valid match). **For real-world queries, we sequentially execute all queries using an algorithm and report the cumulative run time to simulate its overall performance in the long-run.**

6.2 Experimental Results

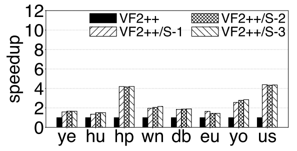
6.2.1 The Space Cost of SUFF. **The space cost for storing all filters is listed in Table 2. For graphs with synthetic queries, we generate filters with pre-defined pattern graphs, so the number of filters is the same for these graphs (Except for WordNet, which has only 5 labels and thus has fewer pattern graphs and filters). The space cost of SUFF for these graphs is then decided by the bit array size. The database size may exceed the original graph size due to the number of filters, but the absolute volume is small compared with modern disk sizes. As for DBpedia, the database is created when executing real-world queries. Most of these queries have only a few vertices, so the number of filters is small. Even though each filter of DBpedia is 8KB, the overall database size is less than 60MB. These results verify the space efficiency of SUFF.**

6.2.2 Overall Performance on Synthetic Queries. **We first investigate the average speedup when using SUFF on all queries, which is shown in Figure 5. In general, algorithms with SUFF run faster, which illustrates its strong pruning power and low overhead. In particular, the acceleration is more obvious on sparse graphs, such as HPRD (5X speedup for QuickSI), WordNet (7X speedup for CFL), and US Patents (6X speedup for DP-iso). The major reason is that, it is less likely to find a match in sparser graphs, leaving more room for pruning**

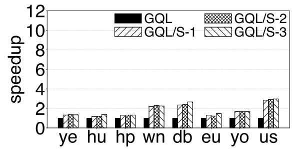
using our technique. On data sets having many labels, such as Yeast, Human, and eu2005, the accelerating effect is limited due to a low chance of finding an available pre-defined filter. Another finding is that, using more filters only brings better acceleration in a few cases and may even harm the overall performance (see DP-iso on US Patents). Thus, setting $k = 1$ would be a safe start when using our technique, which has low overhead and competitive pruning power.



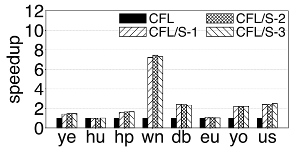
(a) QuickSI.



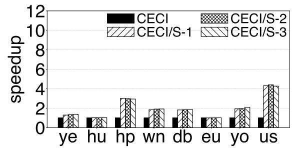
(b) VF2++.



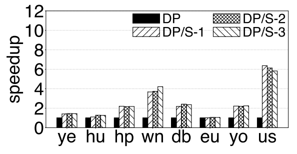
(c) GraphQL.



(d) CFL.



(e) CECI.

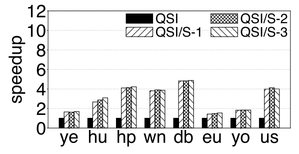


(f) DP-iso.

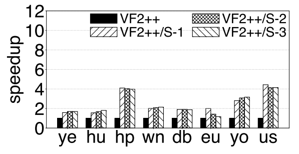
Figure 5: The average speedup on all queries.

6.2.3 Performance on Sparse and Dense Synthetic Queries. **We then investigate the performance on dense and sparse queries, respectively. The results are shown in Figure 6 and Figure 7. It turns out that every method except for VF2++ enjoys diverse acceleration effects on dense and sparse queries, and SUFF is found to have remarkable acceleration effects on both dense queries (over 8X speedup for CFL on Wordnet) and sparse queries (Over 9X speedup for CECI on US Patents). This indicates that one method is usually good at handling a few kinds of query/data graphs, and using SUFF can effectively makeup for its weakness.**

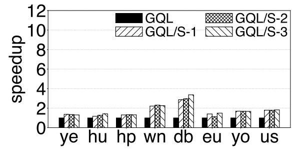
6.2.4 Performance on Large and Small Synthetic Queries. **We then discuss the results on large and small queries, which are shown in Figure 8 and Figure 9. We notice that SUFF can accelerate more on large queries (note that the y-axes of Figure 8 have a wider range than those in other figures). This is because larger queries enjoy more filters, and pruning a branch has more benefit for deeper search trees than for shallower trees. Note that larger queries are harder to process for all algorithms, which highlights the advantage of using**



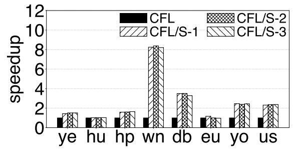
(a) QuickSI.



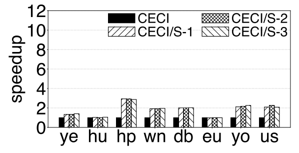
(b) VF2++.



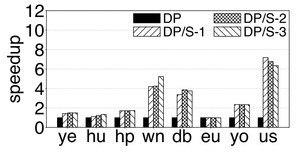
(c) GraphQL.



(d) CFL.

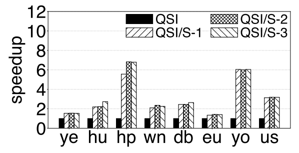


(e) CECI.

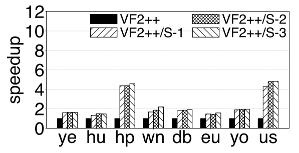


(f) DP-iso.

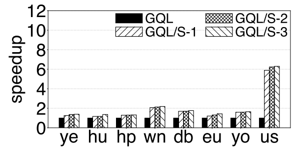
Figure 6: The average speedup on dense queries.



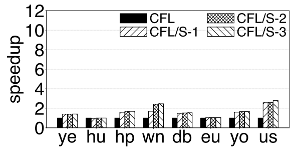
(a) QuickSI.



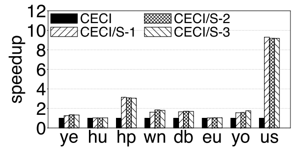
(b) VF2++.



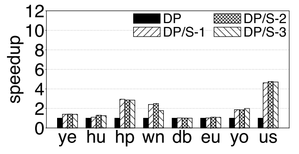
(c) GraphQL.



(d) CFL.



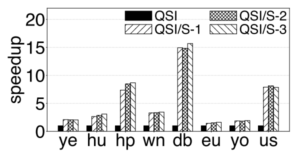
(e) CECI.



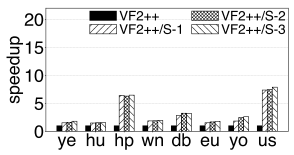
(f) DP-iso.

Figure 7: The average speedup on sparse queries.

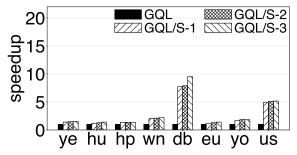
SUFF over existing pruning techniques. In addition, on large queries, we first observed a significant improvement in using more filters (DP-iso on WordNet). However, setting $k = 1$ or 2 is still the most balanced choice considering all results. On small queries, the power of SUFF reduces in most cases, but it still brings more than 7X speedup for CFL on WordNet, showing its effectiveness in these corner cases.



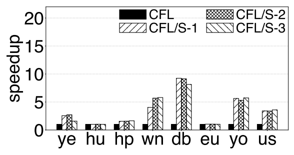
(a) QuickSI.



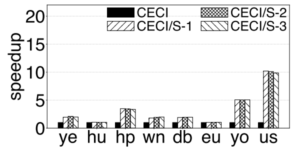
(b) VF2++.



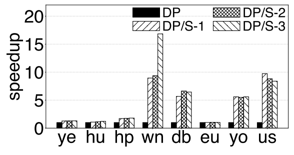
(c) GraphQL.



(d) CFL.



(e) CECI.



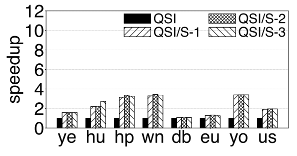
(f) DP-iso.

Figure 8: The average speedup on large queries.

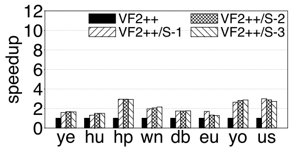
6.2.5 Failing Branch Ratio. **In this part, we illustrate the ratio of branches that SUFF can prune in addition to the original algorithm. The results are shown in Figure 10. In general, fewer failing branches indicate a larger speedup in Figure 5. However, the two numbers are not always directly proportional. The reason is that, if most of the failing branches are at the deep levels, pruning these branches would not decrease the overall running time much. Instead, pruning early branches has better effects of speeding up, such as CFL on WordNet. This result also validates our motivation to apply filters as early as possible.**

6.2.6 Filter Selection Strategy. **In this part, we study the effectiveness of our proposed filter selection strategy. Specifically, we replace algorithm 2 with random selection and report the average speedup in Figure 11. Compared with Figure 5, the average speedup with random filter selection is, in general less than our selection method, meaning that algorithm 2 can effectively pick high-quality filters for use.**

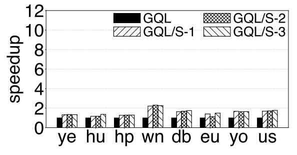
6.2.7 Filter Removal. We also investigate the effect and efficiency of the filter removal mechanism. First, we test the performance



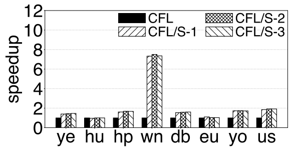
(a) QuickSI.



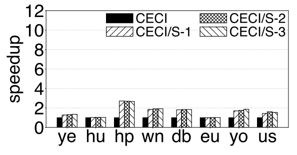
(b) VF2++.



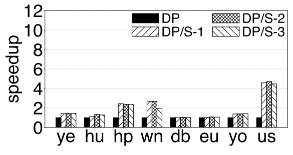
(c) GraphQL.



(d) CFL.

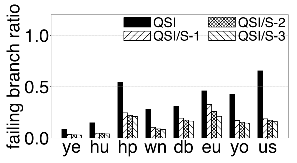


(e) CECI.

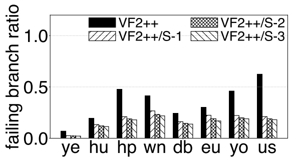


(f) DP-iso.

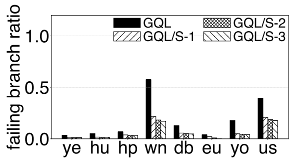
Figure 9: The average speedup on small queries.



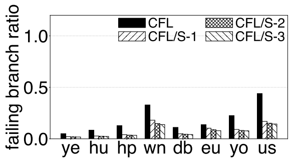
(a) QuickSI.



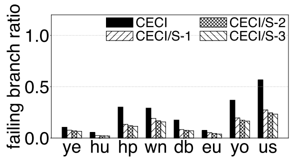
(b) VF2++.



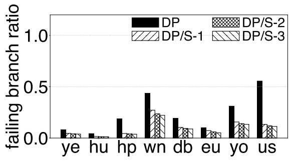
(c) GraphQL.



(d) CFL.

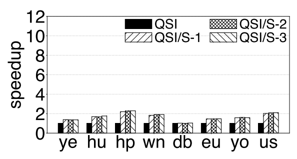


(e) CECI.

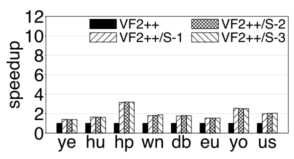


(f) DP-iso.

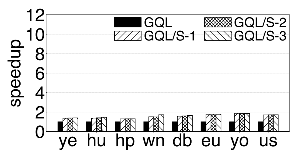
Figure 10: The average failing branch ratio on all queries.



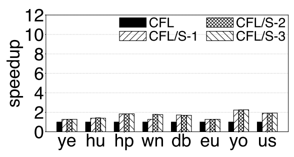
(a) QuickSI.



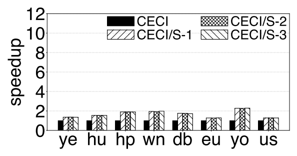
(b) VF2++.



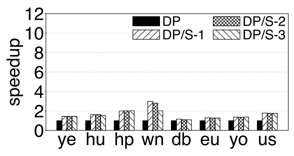
(c) GraphQL.



(d) CFL.



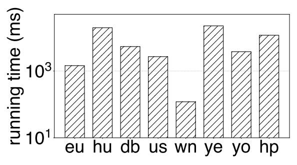
(e) CECI.



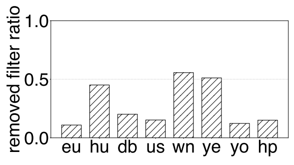
(f) DP-iso.

Figure 11: The average speedup with random filter selection on all queries.

of the filter removal method. the results are shown in Figure 12. It can be seen that, the running time on every data set is quite small (less than a minute), which matches our analysis. In addition, the running time is largely decided by the filter database size. For example, WordNet has only 5 node labels, so it has fewer filter patterns than other data sets. Therefore, it has the shortest filter removal time. The portion of removed filters also varies across all data sets, due to the different distributions of labels and edges. For data sets like WordNet, the algorithm can remove more than half of the filters, which shows the effectiveness of this method.



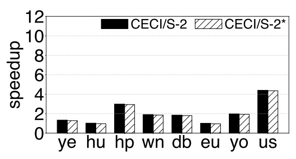
(a) Filter removing time.



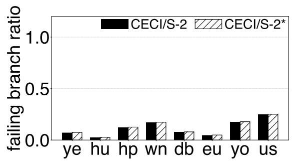
(b) Filter pruning ratio.

Figure 12: The performance of the filter removal algorithm.

Second, we test the influence of removing redundant filters by comparing the running time and failing ratio of CECI before/after running algorithm 3 (marked as CECI/S-2 and CECI/S-2*). The results are shown in Figure 13. In this figure, we set $k = 2$ according to previous results. Clearly, removing redundant filters do not influence the performance of SUFF much. The difference is less than 1% across all data sets.



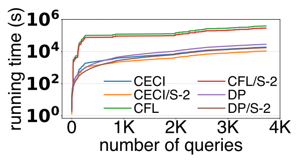
(a) Running time.



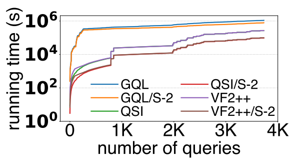
(b) Failing ratio.

Figure 13: The effect of removing redundant filters (CECI).

6.2.8 Performance on Real-World Queries. **Finally, we conduct an experiment for real-world queries on DBpedia. Specifically, we run each algorithm to sequentially execute all real-world queries and count the overall running time. Then we plug in SUFF to run the same queries, and it will build filters for each query and use them in later queries to simulate a long-run matching system. The results are shown in Figure 14. Apparently, SUFF keeps a noticeable acceleration ratio across the entire process. The improvement is consistent with those on HPRD in Figure 9, majorly because knowledge graphs are sparse and have a large number of labels, and real-world SPARQL queries only have a few vertices. We also notice that, though some methods have unstable performance for some of the queries, the acceleration of SUFF is stable and robust.**



(a) CECI, CFL, and DP-iso.



(b) GQL, QSI, and VF2++.

Figure 14: The running time on real-world queries.

6.2.9 Summary of Findings. **In summary, SUFF can help prune much search space in most of the cases in our experiments. While it can accelerate small queries in many cases, SUFF is more powerful on large queries. The pruning power of SUFF depends on the quality of filters, and those built on basic structures already give acceptable performance improvement. We recommend applying SUFF for the "hard" cases that existing approaches do not handle well, that is, when the data graph only has a few labels, or when the query is large in terms of vertices. Filter selection is important for performance guarantee, and filter removal can effectively remove redundant filters from the filter database without influencing the performance much.**

7 RELATED WORK

This work is closely related to subgraph matching and subgraph enumeration. The major difference is that subgraph enumeration focuses on unlabeled graphs.

Subgraph Matching. Subgraph matching has been extensively studied in the literature. Solutions can be divided into two major categories: the backtracking based methods and the join based methods. Representative backtracking based solutions include VF2 [8], QuickSI [30], GraphQL [13], SPath [38], GADDI [37], TurboISO [12], CECI [3], CFL [4], DP-iso [11], and VEQ_M [19]. They use backtracking and recursing to find the matches. It is a common sense that the matching order can significantly influence the efficiency of the backtracking-based algorithm. Thus, various optimizations are proposed recently to find a good matching order which has low cost. For example, in QuickSI the order is decided based on an infrequent-edge-first manner, and CFL uses a path-based ordering method. In DP-iso, dynamic matching order is used where the next vertex to match is picked according to the parent vertex. In addition to the ordering technique, candidate refinement or filtering techniques are also used to reduce the search space. In GraphQL and SPath, neighbors' labels of a vertex are used to filtering invalid matches at early steps. This technique is then extended to a label and degree filtering method, which is widely used in recent solutions CECI, CFL and DP-iso. Besides, other methods are proposed to further refine the candidate vertex sets, such as the compressed path index in CFL, the compact embedding cluster index in CECI, and the failing set pruning in DP-iso. These methods are designed upon different graph properties, making these algorithms suitable for different graphs. The filtering framework proposed in this work is orthogonal to existing filtering techniques since it uses historical matching result instead of artificial rules or assumptions. Thus, it can work together with other optimizations without conflict. The join-based subgraph matching algorithms also attract a large attention recently. Typically, the worst-case optimal join is adapted for subgraph matching in graph systems like EmptyHeaded [1], and Graphflow [17]. In these approaches, a query graph is first decomposed into small parts (units). Since the structure of these parts is simple enough, their matches can be listed directly from the graph or from a pre-processed index. Finally, these partial matches are joined together following a join plan, to obtain the final matches. Common optimizations in these approaches include designing a good unit structure that is both easy to list and easy to join, and reducing the intermediate result size. Compared with the backtracking approaches, join-based approaches enjoy the power of parallel and distributed computations, at the cost of high memory/disk consumption for materializing intermediate results. Therefore, join-based approaches are more suitable for answering small queries (e.g., less than 10 vertices) on large graphs.

Subgraph Enumeration. Compared with subgraph matching, subgraph enumeration faces a much larger search space and result size. Hence, existing approaches commonly utilize the power of parallel or distributed computing to improve efficiency. Kim et al. [20] proposes an I/O-efficient algorithm Dualsim through a dual approach, which executes several sub-plans in parallel. Sun et al. [35] use the Trinity memory cloud to parallelize a join-based algorithm, which employs STwigs as the join unit. Shao et al. [31] parallelize the traditional DFS algorithm using Pregel [25]. They use several pruning rules and the workload-balancing strategy to improve efficiency. Lai et al. [22, 23] investigate the join-based algorithms based on the Crystal structure with MapReduce. They try to reduce the overall I/O cost by introducing different join units and join trees. Gao et al. [10] achieve approximate subgraph enumeration through message passing. They convert the query graph into a DAG, and use Apache Giraph to pass messages between vertices. Qiao et al. [27] propose a framework of vertex-cover-based-compression, which can further reduce the I/O cost.

8 CONCLUSION

In this paper, we study the problem of subgraph matching. Unlike traditional methods that utilize artificial rules or special graph properties, we propose a general filtering framework SUFF which uses historical matching results to accelerate upcoming queries. Specifically, we propose to build a set of filters using partial matches of a historical query q, and then use these filters to prune the search space of a future query q' if q is a subgraph of q'. To strike a balance between the computation overhead and pruning power, we propose a filter utility model, and try to maximize the overall utility while limiting the number of used filters. Though this problem is proved to be NP-hard, we propose an efficient greedy algorithm with approximation bound $1 - \frac{1}{\mathrm{e}}$. Considering that the size of the filter database would grow quickly as more queries are proceed, we propose the concept of filter domination, which enables us to remove unnecessary filters in the database to save space while not harming the pruning effectiveness much. Minimizing the database size in terms of filter counts is also NP-hard, so we propose an efficient greedy algorithm that runs in polynomial time. This algorithm also ensures filter quality across multiple runs. We then conduct extensive experiments on real-world datasets by plugging SUFF into representative subgraph matching algorithms. Results show that SUFF can effectively accelerate these algorithms with a small space and computation overhead.

REFERENCES

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *SIGMOD*.

[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *The Semantic Web*. 722–735.

[3] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *SIGMOD*.

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

[5] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. In *Commun. ACM*.

[6] Diane J Cook and Lawrence B Holder. 2006. *Mining graph data.* John Wiley & Sons.

[7] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *STOC*.

[8] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. In *IEEE transactions on pattern analysis and machine intelligence*.

[9] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *CoNEXT*.

[10] Jun Gao, Chang Zhou, Jiashuai Zhou, and Jeffrey Xu Yu. 2014. Continuous pattern detection over billion-edge graph using distributed framework. In *ICDE*.

[11] 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. In *SIGMOD*.

[12] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*.

[13] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*.

[14] Xun Jian, Yue Wang, Xiayu Lei, Yanyan Shen, and Lei Chen. 2020. DDSL: Efficient Subgraph Listing on Distributed and Dynamic Graphs. In *DASFAA*.

[15] Alpár Juttner and Péter Madarasi. 2018. VF2++: An improved subgraph isomorphism algorithm. In *Discrete Applied Mathematics*.

[16] Sanjay Ram Kairam, Dan J. Wang, and Jure Leskovec. 2012. The Life and Death of Online Groups: Predicting Group Growth and Longevity. In *Proceedings of the*

Fifth ACM International Conference on Web Search and Data Mining.

[17] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *SIGMOD.*

[18] Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations.*

[19] 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 *SIGMOD.*

[20] Hyeonji Kim, Juneyoung Lee, Sourav S Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath HA Jarrah. 2016. DUALSIM: Parallel subgraph enumeration in a massive graph on a single machine. In *SIGMOD.*

[21] Andreas Krause and Carlos Guestrin. 2005. A note on the budgeted maximization of submodular functions. http://snap.stanford.edu/class/cs224w-readings/krause05note.pdf. (2005).

[22] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. In *PVLDB.*

[23] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. In *PVLDB.*

[24] Jure Leskovec, Ajit Singh, and Jon Kleinberg. 2006. Patterns of Influence in a Recommendation Network. In *PAKDD.*

[25] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD.*

[26] Nataša Pržulj. 2007. Biological network comparison using graphlet degree distribution. In *Bioinformatics.*

[27] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph Matching: on Compression and Computation. In *PVLDB.*

[28] Xuguang Ren and Junhu Wang. 2015. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. In *PVLDB.*

[29] Carlos R. Rivero and Hasan M. Jamil. 2017. Efficient and Scalable Labeled Subgraph Matching Using SGMatch. In *KIS.*

[30] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In *PVLDB.*

[31] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel Subgraph Listing in a Large-scale Graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.*

[32] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. 2009. Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics.*

[33] Claus Stadler, Muhammad Saleem, Qaiser Mehmood, Carlos Buil-Aranda, Michel Dumontier, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2022. LSQ 2.0: A linked dataset of SPARQL query logs. *Semantic Web Journal* (2022).

[34] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-Depth Study. In *SIGMOD.*

[35] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. In *PVLDB.*

[36] Douglas Brent West et al. 2001. *Introduction to graph theory.* Prentice hall Upper Saddle River.

[37] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *EDBT.*

[38] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. In *PVLDB.*