Subgraph Matching with Set Similarity in a Large Graph Database

Liang Hong, *Member, IEEE*, Lei Zou, *Member, IEEE*, Xiang Lian, *Member, IEEE*, and Philip S. Yu, *Fellow, IEEE*

Abstract—In real-world graphs such as social networks, Semantic Web and biological networks, each vertex usually contains rich information, which can be modeled by a set of tokens or elements. In this paper, we study a *subgraph matching with set similarity* (SMS^2) query over a large graph database, which retrieves subgraphs that are structurally isomorphic to the query graph, and meanwhile satisfy the condition of vertex pair matching with the (dynamic) weighted set similarity. To efficiently process the SMS^2 query, this paper designs a novel lattice-based index for data graph, and lightweight signatures for both query vertices and data vertices. Based on the index and signatures, we propose an efficient two-phase pruning strategy including set similarity pruning and structure-based pruning, which exploits the unique features of both (dynamic) weighted set similarity and graph topology. We also propose an efficient dominating-set-based subgraph matching algorithm guided by a dominating set selection algorithm to achieve better query performance. Extensive experiments on both real and synthetic datasets demonstrate that our method outperforms state-of-the-art methods by an order of magnitude.

Index Terms—Subgraph matching, set similarity, graph database, index

1 Introduction

With the emergence of many real applications such as social networks, Semantic Web, biological networks, and so on [1], [2], [3], [4], [5], [6], graph databases have been widely used as important tools to model and query complex graph data. Much prior work has extensively studied various types of queries over graphs, in which *subgraph matching* [7], [8], [9], [10] is a fundamental graph query type. Given a query graph Q and a large graph G, a typical subgraph matching query retrieves those subgraphs in G that exactly match with Q in terms of both graph structure and vertex labels [7]. However, in some real graph applications, each vertex often contains a rich set of tokens or elements representing features of the vertex, and the exact matching of vertex labels is sometimes not feasible or practical.

Motivated by the observation above, in this paper, we focus on a variant of the subgraph matching query, called *subgraph matching with set similarity* (SMS^2) query, in which each vertex is associated with a set of elements with *dynamic* weights instead of a single label. The weights of elements are specified by users in different queries according to different application requirements or evolving data. Specifically, given a query graph Q with n vertices u_i $(i = 1, \dots, n)$, the SMS^2 query retrieves all the subgraphs X with n vertices v_j $(j = 1, \dots, n)$ in a large graph G, such that (1) the weighted set similarity between $S(u_i)$ and $S(v_j)$ is larger than a user specified similarity threshold, where $S(u_i)$ and $S(v_j)$ are sets associated with u_i and v_j, respectively; (2) X is structurally isomorphic to Q with u_i mapping to v_j. We discuss the following two examples to demonstrate the usefulness of SMS^2 queries.

Example 1 (Finding papers in DBLP). The DBLP[1] computer science bibliography provides a citation graph G (Fig. 1b) in which vertices represent papers and edges represent citation relations between papers. Each paper contains a set of keywords, in which each keyword is assigned a weight to measure the importance of a keyword with regard to the paper. In reality, a researcher searches for similar papers from DBLP based on both citation relationships and the content similarity of papers [11]. For example, a researcher wants to find papers on subgraph matching that are cited by both social network papers and papers on protein interaction network search. Furthermore, she/he requires papers on protein interaction network search being cited by social network papers. Such query can be modeled as an SMS^2 query, which obtains subgraph matches of the query graph Q (Fig. 1a) in G. Each paper (vertex) in Q and its matching paper in G should have similar set of keywords, and each citation relation (edge) exactly follows the researcher's requirements.

Example 2 (Querying DBpedia). DBPedia extracts entities and facts from Wikipedia and stores them in an RDF graph [12]. As shown in Fig. 2b, in a DBPedia RDF graph G, each entity (i.e., vertex) has an attribute "dbpedia-owl:

- L. Hong is with the School of Information Management, Wuhan University, Wuhan, China. E-mail: hong@whu.edu.cn.
- L. Zou is with the Institute of Computer Science and Technology, Peking University, Beijing, China. E-mail: zoulei@pku.edu.cn.
- X. Lian is with the Department of Computer Science, University of Texas Pan American, Edinburg, TX 78539. E-mail: lianx@utpa.edu.
- P.S. Yu is with the Department of Computer Science, University of Illinois, Chicago, IL 60607, and the Institute for Data Science, Tsinghua University, Beijing, China. E-mail: psyu@uic.edu.

Manuscript received 7 Sept. 2014; revised 11 Dec. 2014; accepted 16 Dec. 2014. Date of publication 11 Jan. 2015; date of current version 3 Aug. 2015. Recommended for acceptance by A. Singh.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2015.2391125

1. http://www.informatik.uni-trier.de/ ley/db/

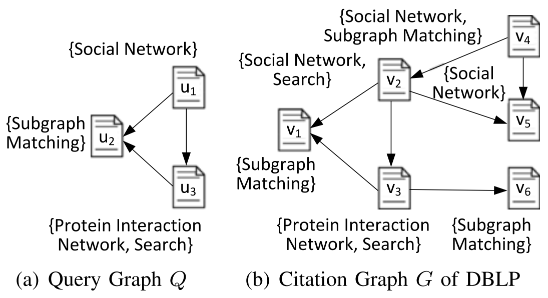


Fig. 1. An example of finding groups of cited papers in DBLP that match with the query citation graph.

abstract" that provides a human-readable description (i.e., a set of words) of the entity, and each edge is a fact that indicates the relationship between entities. Typically, users issue SPARQL queries to find subgraph matches of the query graph (i.e., a graph of connected entities with known attribute values) by specifying exact query criteria. However, in reality, a user may not know (or remember) the exact attribute values or the RDF schema (such as property names). For example, a user wants to find two physicists who both won Nobel prizes and are related to Denmark from DBpedia, while he/she does not know the schema of DBpedia data. In this case, the user can issue an SMS^2 query Q, as shown in Fig. 2a, in which each vertex is described by a short text. The answer to the SMS^2 query Q is Niels Bohr and Aage Bohr, because the subgraph match is structurally isomorphic to Q and the text similarity (measured by the word set similarity) of the matching vertex pairs is high. Interestingly, we find that Niels Bohr is the father of Aage Bohr.

The motivation examples show that SMS^2 queries are very useful in many real-world applications. To the best of our knowledge, no prior work studied the subgraph matching problem under the semantic of structural isomorphism and set similarity with dynamic element weights (called *dynamic weighted set similarity*). Traditional weighted set similarity [13] that focuses on fixed element weight is actually a special case of dynamic weighted set similarity. Due to different matching semantic on vertices (i.e., dynamic weighted set similarity rather than exact label matching), previous techniques on exact or approximate subgraph matching [7], [8], [9], [14], [15], [16] cannot be directly applied to answering SMS^2 queries.

It is challenging to utilize both dynamic weighted set similarity and structural constraints to efficiently answer SMS^2 queries. There are two straightforward methods that answer SMS^2 queries by modifying existing algorithms. The first method conducts the subgraph isomorphism using existing subgraph isomorphism algorithms (e.g., [14], [17]). Then, resulting candidate subgraphs are refined by checking the weighted set similarity between each pair of matching vertices. The second method is in a reverse order, that is, first finding candidate vertices in the data graph that have similar sets to vertices in the query graph by calculating weighted set similarity on-the-fly (as weights change dynamically), which is computationally expensive, and then obtaining matching subgraphs from the candidate vertices. However, these two methods usually incur very high

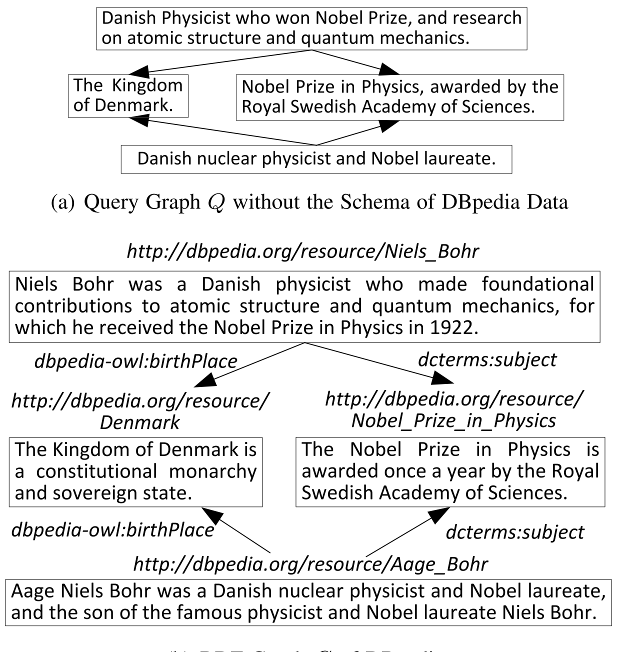

Danish Physicist who won Nobel Prize, and research on atomic structure and quantum mechanics.
The Kingdom of Denmark.
Nobel Prize in Physics, awarded by the Royal Swedish Academy of Sciences.
Danish nuclear physicist and Nobel laureate.
(a) Query Graph Q without the Schema of DBpedia Data
http://dbpedia.org/resource/Niels_Bohr
Niels Bohr was a Danish physicist who made foundational contributions to atomic structure and quantum mechanics, for which he received the Nobel Prize in Physics in 1922.
dbpedia-owl:birthPlace
dcterms:subject
http://dbpedia.org/resource/Denmark
http://dbpedia.org/resource/Nobel_Prize_in_Physics
The Kingdom of Denmark is a constitutional monarchy and sovereign state.
The Nobel Prize in Physics is awarded once a year by the Royal Swedish Academy of Sciences.
dbpedia-owl:birthPlace
dcterms:subject
http://dbpedia.org/resource/Aage_Bohr
Aage Niels Bohr was a Danish nuclear physicist and Nobel laureate, and the son of the famous physicist and Nobel laureate Niels Bohr.
(b) RDF Graph G of DBpedia

Fig. 2. An example of querying subgraph matches from the RDF graph of DBpedia.

query cost, especially for a large graph database. This is because the first method ignores the weighted set similarity constraints, whereas the second one ignores the structural information when filtering candidate results.

Due to the inefficiency of existing methods, we propose an efficient SMS^2 query processing approach in this paper. Our approach adopts a "filter-and-refine" framework, which exploits unique features of both graph topology and dynamic weighted set similarity. In the filtering phase, we build a *lattice-based index* over frequent patterns of *element set*s of vertices in data graph G. Then, data vertices are encoded into signatures, and organized into *signature bucket*s. We design an efficient two-phase pruning strategy based on the lattice-based index and signature buckets to reduce the SMS^2 search space. In the refinement phase, we propose a *dominating set*[2] (DS)-based subgraph matching algorithm to find *subgraph matches with set similarity* (defined in Definition 1). A dominating set selection method is proposed to select a cost-efficient dominating set of the query graph. In summary, we make the following contributions:

1) We design a novel strategy to efficiently process SMS^2 queries. An inverted pattern lattice based indexing and a structural signature-based locality sensitive hashing (LSH) are first constructed offline. During the online phase, a set of pruning techniques facilitated by the offline data structures are introduced and integrated together to greatly reduce the search space of SMS^2 queries.

2) We propose set similarity pruning techniques (Section 4) that utilize a novel *inverted pattern lattice* over the element sets of data vertices to evaluate dynamic weighted set similarity. It introduces an upper bound on the dynamic weighted

2. In graph theory, a dominating set for a graph $Q = (V, E)$ is a subset $DS(Q)$ of $V(Q)$ such that every vertex of Q is either in $DS(Q)$, or adjacent to some vertex in $DS(Q)$.

similarity measure to apply the *anti-monotone principle* to achieve high pruning power.

3) We propose structure-based pruning techniques (Section 5) that explore a novel structural-signature-based data structure, where the signature is designed to capture the set and neighborhood information. An *aggregate dominance principle* is devised to guide the pruning.

4) Instead of directly querying and verifying the candidates of all the vertices in the query graph, we design an efficient algorithm (Section 6) to perform subgraph matching based on the dominating set of query graph. When filling up the remaining (non-dominating) vertices of the graph, a *distance preservation principle* is devised to prune candidate vertices that do not preserve the distance to dominating vertices.

5) Last but not least, we demonstrate through extensive experiments that our approach can effectively and efficiently answer the SMS^2 queries in a large graph database.

2 Related Work

Exact subgraph matching query requires that all the vertices and edges are matched exactly. The Ullmann's subgraph isomorphism method [17] and VF2 [14] algorithm do not utilize any index structure, thus they are usually costly for large graphs. GADDI [18] is a structure distance based subgraph matching algorithm in a large graph. Zhao and Han [8] investigated the SPath algorithm, which utilizes shortest paths around the vertex as basic index units. Cheng et al. [2] proposed a new two-step R-join algorithm to efficiently find matching graph patterns from a large graph. Zou et al. [9] proposed a distance-based multi-way join algorithm for answering pattern match queries over a large graph. Shang et al. [19] proposed QuickSI algorithm for subgraph isomorphism optimized by choosing an search order based on some features of graphs. SING [20] is a novel indexing system for subgraph isomorphism in a large scale graph. GraphQL [21] is a query language for graph databases which supports graphs as the basic unit of information. Sun et al. [10] utilized graph exploration and parallel computing to process subgraph matching query on a billion node graph. Recently, an efficient and robust subgraph isomorphism algorithm Turbo$_{ISO}$ [15] was proposed. RINQ [22] and GRAAL [23] are graph alignment algorithms for biological networks, which can be used to solve isomorphism problems. To solve subgraph isomorphism problems, graph alignment algorithms introduce additional cost as they should first find candidate subgraphs of similar size from the large data graph. In addition, existing exact subgraph matching and graph alignment algorithms do not consider weighted set similarity on vertices, which will cause high post-processing cost of set similarity computation.

Approximate subgraph matching query usually concerns the structure information and allows some of the vertices or edges not being matched exactly. Closure-tree [16] is the first graph index that supports both subgraph queries and graph similarity queries. SAGA [24] is an approximate subgraph matching technique that finds subgraphs in the database that are similar to the query, allowing for node mismatches, node gaps, and graph structural differences. Torque [4] is a topology-free querying tool of protein interaction network. Torque does not require precise information on interaction pattern of the query. TALE [7] is an index-based method for approximate subgraph matching which allows node mismatches, and node/edge insertions and deletions. Our work differs from the approximate subgraph matching problems in that no node/edge mismatches are allowed, and the matching vertices should have similar sets.

Recently, several novel subgraph similarity search problems have been investigated. Ma et. al [25] studied the problem of *graph simulation* by enforcing duality and locality conditions on subgraph matches. NeMa [26] focuses on the subgraph matching queries that satisfy the following two conditions (1) many-to-one subgraph matching with a cost function, and (2) label similarity of matching vertices. S^4 system [27] finds the subgraphs with identical same structure and semantically similar entities of query subgraph. SMS^2 query differs from the above problems in that it considers both one-to-one structural isomorphism and dynamic set similarity of matching vertices. Zou et al. [28] proposed a top-k subgraph matching problem that considers the similarity between objects associated with two matching vertices. This work assumes that all vertex similarities are given, and does not exploit set similarity pruning techniques to optimize subgraph matching performance.

As for the weighted set similarity query, Hadjieleftheriou et al. [29] proposed various index structures and algorithms. Recently, a *Heaviest First* strategy [13] has been proposed for efficiently answering all-match weighted string similarity query. However, dynamic element weights (i.e. query dependent weights) in SMS^2 queries make most of existing index structures and query processing techniques for weighted set similarity inefficient, or even infeasible. The reason is that these methods rely on element canonicalization according to fixed weights, while elements with dynamic weights cannot be canonicalized in advance.

3 Preliminaries

3.1 Problem Definition

In this subsection, we formally define our problem of subgraph matching with set similarity query. Specifically, we consider a large graph G represented by $\langle V(G), E(G)\rangle$, where $V(G)$ is a set of vertices, $E(G)$ is a set of edges. Each vertex $v \in V(G)$ is associated with a set, $S(v)$, of elements. Query graph Q is represented by $\langle V(Q), E(Q)\rangle$. The set of element domain is denoted by $\mathcal{U}$, in which each element a has a weight $W(a)$ to indicate the importance of a. Note that, weights can change dynamically in different queries due to varying requirements or evolving data in real applications (e.g., Examples 1 and 2).

Definition 1 (Subgraph Match with Set Similarity). *For a query graph Q with n vertices $(u_1, \ldots, u_n)$, a data graph G, and a user-specified similarity threshold τ, a* subgraph match *of Q is a subgraph X of G containing n vertices $(v_1, \ldots, v_n)$ of $V(G)$, iff the following conditions hold:*

1) *There is a mapping function f, for each u_i in $V(Q)$ and $v_j \in V(G)$ $(1 \le i \le n,\ 1 \le j \le n)$, it holds that $f(u_i) = v_j$;*

2) *$sim(S(u_i), S(v_j)) \geq \tau$, where $S(u_i)$ and $S(v_j)$ are the sets associated with u_i and v_j, respectively, and $sim(S(u_i), S(v_j))$ outputs a set similarity score between $S(u_i)$ and $S(v_j)$;*
3) *For any edge $(u_i, u_k) \in E(Q)$, there is $(f(u_i), f(u_k)) \in E(G)$ $(1 \leq k \leq n)$.*

Since the subgraph match with set similarity is independent of directed or undirected edges, the proposed techniques can be applied to both directed and undirected graph. We define the SMS^2 query as follows:

Definition 2 (SMS^2 Query). *Given a query graph Q and a data graph G, a* subgraph matching with set similarity *query retrieves all subgraph matches of Q in graph G under the semantic of the set similarity.*

Note that, the choice of the similarity function $sim(.,.)$ highly depends on the application domain. In this paper, we choose *weighted Jaccard similarity*, which is one of the most widely used similarity measures.

Definition 3 (Weighted Jaccard Similarity). *Given element sets $S(u)$ and $S(v)$ of vertices u and v, the* weighted Jaccard similarity *between $S(u)$ and $S(v)$ is*

$$sim(S(u), S(v)) = \frac{\sum_{a \in S(u) \cap S(v)} W(a)}{\sum_{a \in S(u) \cup S(v)} W(a)}, \quad (1)$$

where $W(a)$ is the weight of element a, $W(a) \geq 0$.

In Definition 3, when $W(a) = 1$ for each element a, the weighted Jaccard similarity is exactly the classical Jaccard similarity [30]. As mentioned in [30], some other popular similarity measures such as *cosine similarity*, *Hamming distance* and *overlap similarity* can be converted to Jaccard similarity. Therefore, given another similarity measure (e.g., cosine similarity, Hamming distance and overlap similarity) and a threshold α, we can transform it into Jaccard similarity with a corresponding threshold τ. Then, we process the SMS^2 query using a constant lower bound to τ. Finally, we verify each candidate by the original similarity measure (detailed in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2015.2391125).

In real applications, the weight of each element a can be specified by the query issuer or weighting methods such as TF/IDF [13]. For instance, in Example 1, the weight of each keyword represents the correlation between the keyword and the paper, which is specified by researchers. In Example 2, each token in DBpedia can be assigned with a TF/IDF weight, which changes dynamically due to evolving data.

3.2 Framework

In this subsection, we present a filter-and-refine framework for our proposed approaches, which includes offline processing and online processing, as shown in Fig. 3.

Offline processing. We construct a novel *inverted pattern lattice* to facilitate efficient pruning based on the set similarity. Since the dynamic weight of each element makes existing indices inefficient for answering SMS^2 queries, we need to design a novel index for SMS^2 query. Motivated by the anti-monotone property of the lattice structure (see details in Section 4), we mine *frequent pattern*s (defined in Definition 5) from element sets of vertices in the data graph G, and organize them into a lattice. We store data vertices in the inverted list for each frequent pattern P, if P is contained in the element sets of these vertices. The lattice together with the inverted lists is called *inverted pattern lattice*, which can greatly reduce the cost of dynamic weighted set similarity search. To support structure-based pruning, we encode each query vertex and data vertex into a *query signature* and a *data signature* respectively by considering both the set and topology information, and hash all the data signatures into *signature bucket*s.

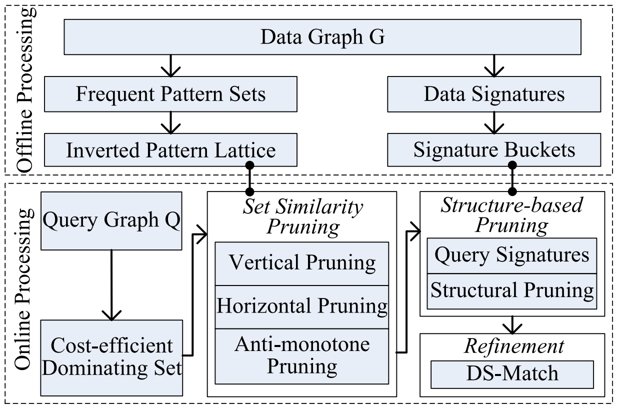


Fig. 3. Framework for SMS^2 query processing.

Online processing. We propose finding a cost-efficient *dominating set* (defined in Section 6) of the query graph Q, and only search candidates for vertices in the dominating set. Note that, different dominating sets will lead to different query performances. Thus, we propose a dominating set selection algorithm to select a cost-efficient dominating set $DS(Q)$ of query graph Q.

The dominating-set-based subgraph matching is motivated by two observations: (1) finding candidates in SMS^2 queries is much more expensive than that in typical subgraph search, because set similarity calculation is more costly than vertex label matching. As a result, the filtering cost can be reduced by searching dominating vertices of $V(Q)$ rather than all query vertices. (2) Some query vertices may have a large amount of candidate vertices, which leads to many unnecessary intermediate results during subgraph matching. Therefore, the subgraph matching cost can also be reduced by decreasing the size of intermediate results.

For each vertex $u \in DS(Q)$, we propose a two-phase pruning strategy, including set similarity pruning and structure-based pruning. The set similarity pruning includes anti-monotone pruning, vertical pruning, and horizontal pruning, which are based on our proposed inverted pattern lattice (Section 4). Based on the signature buckets, we also propose the structure-based pruning technique by utilizing novel vertex signatures (Section 5).

After the pruning, we propose an efficient *DS-Match* subgraph matching algorithm to obtain subgraph matches of Q based on candidates of dominating vertices in $DS(Q)$ (Section 6). DS-match utilizes topological relations between

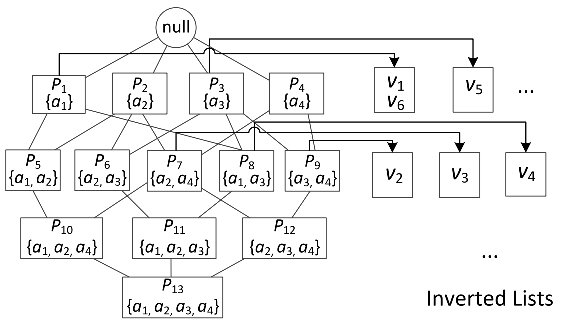


Fig. 4. Example of inverted pattern lattice including a lattice of frequent patterns and inverted lists.

dominating vertices and non-dominating vertices to reduce the scale of intermediate results during subgraph matching, and therefore reduces the matching cost.

4 Set Similarity Pruning

For a vertex u in a dominating set of query graph Q, we need to find its candidate vertices in graph G. Let us recall the definition of SMS^2 query in Definition 1. If a vertex v in graph G match with u, $sim(S(u), S(v)) > \tau$ holds. This section concentrates on finding candidate vertices v of u such that $sim(S(u), S(v)) > \tau$. How to select a cost-efficient dominating set will be introduced in Section 6.2.

As discussed in Section 2, existing indices relying on element canonicalization are not suitable for SMS^2 queries due to dynamic weights of elements. Nevertheless, we note that the *inclusion relation* between two sets does not change even if element weights vary dynamically. For two element sets $S(v)$ and $S(v')$ of vertex v and v' respectively, if $S(v) \subseteq S(v')$, the relationship of $S(v)$ being a subset of $S(v')$ is called inclusion relation. Based on inclusion relation, we derive the following upper bound.

Definition 4 (AS Upper Bound). *Given a query vertex u's set $S(u)$ and a data vertex v's set $S(v)$, an Anti-monotone Similarity (AS) upper bound is*

$$UB(S(u), S(v)) = \frac{\sum_{a\in S(u)} W(a)}{\sum_{a\in S(u)\cup S(v)} W(a)} \geq sim(S(u), S(v)), \tag{2}$$

where $W(a)$ denotes the weight assigned to element a, and $sim(,)$ is given by Equation (1).

Since $\sum_{a\in S(u)} W(a)$ does not change once the query is given, AS upper bound is anti-monotone with regard to $S(v)$. That is, for any set $S(v) \subseteq S(v')$, if $UB(S(u), S(v)) < \tau$, then $UB(S(u), S(v')) < \tau$.

Apparently, the anti-monotone property of AS upper bound enables us to prune vertices based on inclusion relations. However, inclusion relations between element sets of data vertices are few. In contrast, since most of element sets contain *frequent pattern*s, inclusion relations between element sets and frequent patterns are numerous. Motivated by this observation, we mine frequent patterns from element sets of all the data vertices, and design a novel index structure named *inverted pattern lattice* to organize frequent patterns. The lattice-based index enables efficient anti-monotone pruning, and therefore is suitable for set similarity search with dynamic weights.

Here, we recall the definition of the frequent pattern [31].

Definition 5 (Frequent Pattern). *Let $\mathcal{U}$ be the set of distinct elements in $V(G)$. A pattern P is a set of elements in $\mathcal{U}$, i.e., $P \subseteq \mathcal{U}$. If an element set $S(v)$ contains all the elements of a pattern P, then we say $S(v)$ supports P and $S(v)$ is a supporting element set of P. The support of P, denoted by $supp(P)$, is the number of element sets that support P. If $supp(P)$ is larger than a user-specified threshold $minsup$, then P is called a* frequent pattern.

The computation of support of pattern P (i.e., $supp(P)$) can be referred to [31].

4.1 Inverted Pattern Lattice Construction

To build an inverted pattern lattice, we first mine frequent patterns from element sets of all vertices in data graph G. Then, we organize these frequent patterns into a lattice. In the lattice, each intermediate node P is a frequent pattern, which is a subset of all its descendant nodes. We denote the frequent pattern having k elements as a k-frequent pattern. To ensure the completeness of the indexing, the one-frequent patterns (i.e., all elements in the universe $\mathcal{U}$) are indexed in the first (top) level of the lattice. For each k-frequent pattern ($k > 1$) P in the lattice, we insert each vertex v's element set $S(v)$ into the inverted list of P (denoted as $L(P)$), iff $S(v)$ supports P. Note that, an element set $S(v)$ may support multiple frequent patterns in the lattice. We insert $S(v)$ to the inverted lists of these frequent patterns, respectively.

Example 3. Fig. 4 is an example of inverted pattern lattice built by data vertices in Fig. 1b. The elements a_1, a_2, a_3 and a_4 correspond to keywords "Subgraph Matching", "Protein Interaction Network", "Social Network", and "Search", respectively. Then, $v_1 = \{a_1\}$, $v_2 = \{a_3, a_4\}$, $v_3 = \{a_2, a_4\}$, $v_4 = \{a_1, a_3\}$, $v_5 = \{a_3\}$, $v_6 = \{a_1\}$.

4.2 Pruning Techniques

4.2.1 Anti-Monotone Pruning

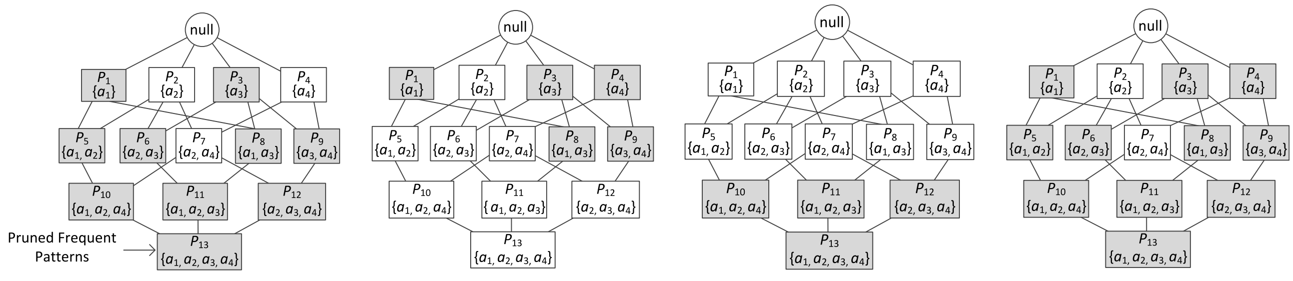
Considering the anti-monotone property of AS upper bound and the characteristics of inverted lattice pattern , we have the following theorem.

Theorem 1. *Given a query vertex u, for each accessed frequent pattern P in the inverted pattern lattice, if $UB(S(u), P) < \tau$, all vertices in the inverted list $L(P)$ and $L(P')$ can be safely pruned, where P' is a descendant node of P in the lattice.*

Proof. For each element set $S(v)$ in the inverted list of P, since $P \subseteq S(v)$, $UB(S(u), S(v)) < \tau$ according to the anti-monotone property of AS upper bound. Similarly, for any descendant node P' of P, since $P' \subset P$, $UB(S(u), P')$ will be also less than τ. The theorem can be proved. □

Based on the theorem above, we can efficiently prune frequent patterns in the inverted pattern lattices regardless of dynamic weights of elements.

Example 4. Considering the query vertex u_3 in Fig. 1a, u_3's element set $S(u_3)=\{a_2, a_4\}$. Assume $W(a_1) = 0.5$, $W(a_2) = 0.4$, $W(a_3) = 0.5$, $W(a_4) = 0.2$, and the similarity threshold



(a) Anti-monotone Pruning (b) Vertical Pruning (c) Horizontal Pruning (d) Final Results

Fig. 5. An example of set similarity pruning on inverted pattern lattice.

$\tau = 0.6$. As shown in Fig. 5a, since $UB(S(u_3), P_6) = 0.55 < 0.6$, P_6 and all its descendant nodes in the lattice, i.e., the gray nodes, can be safely pruned. In the same way, we can prune P_1, P_3 and all their descendent nodes. All vertices in the inverted lists of these patterns can be pruned safely.

4.2.2 Vertical Pruning

Vertical pruning is based on the *prefix filtering* principle [32]: if two canonicalized sets are similar, the prefixes of these two sets should overlap with each other, as otherwise these two sets will not have enough common elements.

We canonicalize all the elements in query set $S(u)$ in a descending order of their weights during online processing. The first p elements in the canonicalized set $S(u)$ is denoted as the p-prefix of $S(u)$. We find the maximum prefix length p such that if $S(u)$ and $S(v)$ have no overlap in p-prefix, $S(v)$ can be safely pruned, because they do not have enough overlap to meet the similarity threshold [32].

To find p-prefix of $S(u)$, each time we remove the element with the largest weight from $S(u)$, we check whether the remaining set $S'(u)$ meets the similarity threshold with $S(u)$. We denote L_1-norm of $S(u)$ as $\|S(u)\|_1 = \sum_{a \in S(u)} W(a)$. If $\|S'(u)\|_1 < \tau \times \|S(u)\|_1$, the removal stops. The value of p is equal to $|S'(u)| - 1$, where $|S'(u)|$ is the number of elements in $S'(u)$. For any set $S(v)$ that does not contain the elements in $S(u)$'s p-prefix, we have $sim(S(u), S(v)) \le \frac{\|S'(u)\|_1}{\|S(u)\|_1} < \tau$, so $S(u)$ and $S(v)$ will not meet the set similarity threshold.

Theorem 2. *Given a query set $S(u)$ and a frequent pattern P in the lattice, if P is not a one-frequent pattern (or its descendant) in $S(u)$'s p-prefix, all vertices in the inverted list $L(P)$ can be safely pruned.*

Proof. For each vertex v in $L(P)$, $P \subseteq S(v)$. According to prefix filtering principle and Theorem 1, all vertices in $L(P)$ can be pruned. □

After we find the p-prefix of $S(u)$, we only need to access all the descendent nodes of the corresponding 1-frequent patterns in p-prefix in a vertical manner.

Example 5. For the query vertex u_3 with $S(u_3) = \{a_2, a_4\}$, we can determine the p-prefix of $S(u_3)$ is $\{a_2\}$. As shown in Fig. 5b, we only need to access P_2 and its all descendent nodes in the lattice. All vertices in the inverted lists of other frequent patterns can be pruned.

4.2.3 Horizontal Pruning

Intuitively, a query element set $S(u)$ cannot be similar to a frequent pattern of a large size set or a frequent pattern of a very small size. The size of a frequent pattern P (denoted by $|P|$) is the number of elements in P. In the inverted pattern lattice, each frequent pattern P is a subset of data vertices (i.e., element sets) in P's inverted list. Suppose we can find the length upper bound for $S(u)$ (denoted by $LU(u)$). If the size of P is larger than $LU(u)$, (i.e., the sizes of all data vertices in P's inverted list are larger than $LU(u)$) then P and its inverted list can be pruned.

Due to dynamic element weights, we need to find $S(u)$'s length interval on the fly. We find $LU(u)$ by adding elements in $(\mathcal{U} - S(u))$ to $S(u)$ in an increasing order of their weights. Each time an element is added, a new set $S'(u)$ is formed. We calculate the similarity value between $S(u)$ and $S'(u)$. If $sim(S(u), S'(u)) \ge \tau$ holds, we continue to add elements to $S'(u)$. Otherwise, the upper bound $LU(u)$ equals to $|S'(u)| - 1$.

Note that, frequent patterns at the same level of the inverted pattern lattice have the same size, and the size of frequent patterns at Level k equals to k. Thus, after obtaining the length upper bound $LU(u)$ of $S(u)$, we can determine that the horizontal upper bound equals to $LU(u)$. All frequent patterns under Level $LU(u)$ will be pruned.

Example 6. Considering the query vertex u_3 with $S(u_3) = \{a_2, a_4\}$, and $\mathcal{U} = \{a_1, a_2, a_3, a_4\}$. To find $S(u_3)$'s length upper bound $LU(u)$, we add a_1 (or a_3) to $S(u_3)$ to form $S'(u_3)$. It is clear that $sim(S(u_3), S'(u_3)) < 0.6$. Therefore, $LU(u_3)$ is 2. As shown in Fig. 5c, all frequent patterns below Level 2 can be pruned.

4.2.4 Putting All Pruning Techniques Together

In this subsection, we apply all the set similarity pruning techniques and obtain candidates for a query vertex.

For each query vertex u, we first use vertical pruning and horizontal pruning to filter out false positive patterns and jointly determine the nodes (i.e., frequent patterns) that should be accessed in the inverted pattern lattice. Then, we traverse them in a breadth-first manner. For each accessed node P in the lattice, we check whether $UB(S(u), P)$ is less than τ. If yes, P and its descendant nodes are pruned safely. As shown in Fig. 5d, the grey nodes can be pruned after all pruning techniques above have been applied. Assume that P_1, $P_2, \ldots,$ and P_k are the remaining nodes (i.e., white nodes) that cannot be pruned, the candidate set of u is a subset of $\bigcup_{i=1}^{k} L(P_i)$. Note that, all the child nodes of P_i ($1 \leq i \leq k$), denoted by PN, have been pruned based on AS upper bound. The candidate set $C(u)$ of u can be obtained by Equation (3)

$$C(u) = \bigcup_{i=1}^{k} L(P_i) - \bigcup_{P \in PN} L(P). \tag{3}$$

To increase the query performance, the lattice of frequent patterns resides in the main memory, and the inverted lists of frequent patterns are on the disk.

4.3 Optimization for Inverted Pattern Lattice

In this section, we propose two techniques to optimize the query time and space cost of the inverted pattern lattice.

4.3.1 Vertex Insertion Criterion

For the element set, $S(v)$, of a data vertex v, suppose $S(v)$ supports frequent patterns $P_1, \ldots,$ and P_n, as mentioned above, we insert v into inverted lists $L(P_1), \ldots,$ and $L(P_n)$. However, it is not necessary to insert v to all the inverted lists based on our proposed anti-monotone pruning.

Suppose frequent patterns P' and P are both supported by $S(v)$, and $P' \subset P$. Then, for any query set $S(u)$, we have $UB(S(u), P') > UB(S(u), P)$. Therefore, if $S(v)$ in P's inverted list cannot be pruned by $UB(S(u), P)$ (i.e., $UB(S(u), P) > \tau$), it will definitely not be pruned by $UB(S(u), P')$, as $UB(S(u), P')$ will also be larger than τ. In this case, we only need to insert vertex v into P's inverted list, instead of P''s inverted list.

In summary, let $\{P_1, \ldots, P_n\}$ be the set of frequent patterns supported by $S(v)$, and suppose there are k ($k \leq n$) *paths* formed by $\{P_1, \ldots, P_n\}$. A path is a sequence of connected nodes (i.e., frequent patterns) from higher level to lower level without cycle in the inverted pattern lattice. We only insert v into the inverted lists of the frequent patterns that are supported by $S(v)$ at the lowest level of each path. In this way, the space cost of inverted lists can be reduced.

4.3.2 Frequent Pattern Selection

Since the number of frequent patterns is large, if we index all the frequent patterns in the inverted pattern lattice, the lattice cannot be fitted to memory. As mentioned in Section 4.2.1, frequent patterns with more elements output tighter AS upper bound. Therefore, we should select frequent patterns with numerous elements, so that the querying time can be reduces by pruning more patterns. In addition, we should also guarantee the representativeness of each frequent pattern. Motivated by the observations, we mine *closed frequent patterns* [31] from the sets of vertices.

Definition 6 (Closed Frequent Pattern). *A frequent pattern P is closed if there is no pattern P' such that $P \subset P'$ and $supp(P) = supp(P')$.*

Note that, the size of lattice is controlled by the support threshold $minsup$. Given a memory size M, we need to determine the value of $minsup$ to ensure that the lattice fits into the main memory. Suppose the average memory occupation of each pattern is Z, there are at most $\frac{M}{Z}$ patterns are allowed to be resident in the main memory. In most of real applications, $\frac{M}{Z}$ is no less than the size of the set of one-frequent patterns (i.e., patterns containing only one element) F_1 (denoted by $|F_1|$). We sort all the closed k-frequent patterns P ($k \geq 2$) in a descending order of $supp(P)$, and select the top-($\frac{M}{Z} - |F_1|$) closed k-frequent patterns. We build the lattice by all the one-frequent patterns and the selected closed k-frequent patterns ($k \geq 2$).

5 Structure-Based Pruning

A matching subgraph should not only have its vertices (element sets) similar to corresponding query verices, but also preserve the same structure as Q. Thus, in this section, we design lightweight signatures for both query vertices and data vertices to further filter the candidates after set similarity pruning by structural constraints.

5.1 Structural Signatures

We define two distinct types of structural signature, namely *query signature* $Sig(u)$ and *data signature* $Sig(v)$ for each query vertex u and data vertex v, respectively. To encode structural information, $Sig(u)/Sig(v)$ should contain the element information of both u/v and its surrounding vertices. Since the query graph is usually small, we generate accurate query signatures by encoding each neighbor vertex separately. On the contrary, the data graph is much larger than the query graph, so the aggregation of neighbor vertices can save a lot of space. The pruning cost can be also reduced due to limited number of data signatures.

Specifically, we first sort elements in element sets $S(u)$ and $S(v)$ according to a predefined order (e.g., alphabetic order). Based on the sorted sets, we encode the element set $S(u)$ by a bit vector, denoted by $BV(u)$, for the former part of $Sig(u)$. In particular, each position $BV(u)[i]$ in the vector corresponds to one element a_i, where $1 \leq i \leq |\mathcal{U}|$ and $|\mathcal{U}|$ is the total number of elements in the universe $\mathcal{U}$. If an element a_j belongs to set $S(u)$, then in bit vector $BV(u)$, we have $BV(u)[j] = 1$; otherwise (if $a_j \notin S(u)$), $BV(u)[j] = 0$ holds. Similarly, $S(v)$ is also encoded using the above technique. For the latter part of $Sig(u)$ and $Sig(v)$ (i.e., encoding surrounding vertices), we propose two different encoding techniques for $Sig(u)$ and $Sig(v)$, respectively. The difference is that, we encode every neighbor vertex separately in $Sig(u)$, but aggregate all neighbor vertices in $Sig(v)$. We formally define the query signature and the data signature as follows:

Definition 7 (Query Signature). *Given a vertex u with m adjacent neighbor vertices u_i ($i = 1, \ldots, m$) in a query graph Q, the query signature $Sig(u)$ of vertex u is given by a set of bit vectors, that is, $Sig(u) = \{BV(u), BV(u_1), \ldots, BV(u_m)\}$, where $BV(u)$ and $BV(u_i)$ are the bit vectors that encode elements in set $S(u)$ and $S(u_i)$, respectively.*

Definition 8 (Data Signature). *Given a vertex v with n adjacent neighbor vertices v_i ($i = 1, \ldots, n$) in a data graph G, the data signature, $Sig(v)$, of vertex v is given by: $Sig(v) = [BV(v), \vee_{i=1}^{n} BV(v_i)]$, where $\vee$ is a bitwise OR operator, $BV(v)$ is the bit vector associated with v, and $\vee_{i=1}^{n} BV(v_i)$ (denoted as $BV_{\cup}(v)$) is called a* union bit vector, *which equals to bitwise-OR over all bit vectors of v's one-hop neighbors.*

5.2 Signature-Based LSH

To enable efficient pruning based on structural information, we use a set of *Locality Sensitive Hashing* [33] hash functions to hash each data signature $Sig(v)$ into a *signature bucket*, which is defined as follows.

Definition 9. *A signature bucket is a slot of hash table that stores a set of data signatures with the same hash value.*

Since the probability of collision (i.e., the same hash value) is much higher for signatures that are similar to each other than for those that are dissimilar, the maximum hamming distance of data signatures in each bucket can be minimized. The choice of LSH hash functions can be referred to [33]. In addition, we store the *bucket signature* $Sig(B)$, which is formed by ORing all data signatures in each bucket B. That is $Sig(B) = [\vee BV(v_t), \vee BV_{\cup}(v_t)]$, where $t = 1, \ldots, n$, and n is the number of signatures in B.

5.3 Structural Pruning

Based on the SMS^2 query definition (Definition 1), a data vertex v can be pruned if the similarity between $BV(u)$ and $BV(v)$ is smaller than τ, or there does not exist $BV(v_j)$ that satisfies the similarity constraint with $BV(u_i)$, where v_j $(j = 1, \ldots, n)$ and u_i $(i = 1, \ldots, m)$ are one-hop neighbors of v and u, respectively. We define the similarity between $BV(u)$ and $BV(v)$ as follows, which is analogous to the weighted Jaccard similarity (Definition 3).

Definition 10. *Given bit vectors $BV(u)$ and $BV(v)$, the similarity between $BV(u)$ and $BV(v)$ is:*

$$sim(BV(u), BV(v)) = \frac{\sum_{a\in BV(u)\wedge BV(v)} W(a)}{\sum_{a\in BV(u)\vee BV(v)} W(a)}, \tag{4}$$

where $\wedge$ is a bitwise AND operator and $\vee$ is a bitwise OR operator, $a \in (BV(u) \wedge BV(v))$ means the bit corresponding to element a is 1, $W(a)$ is the assigned weight of a.

For each $BV(u_i)$, we need to determine whether there exists a $BV(v_j)$ so that $sim(BV(u_i), BV(v_j)) \geq \tau$ holds. To this end, we estimate the *union similarity upper bound* between $BV(u_i)$ and $BV_{\cup}(v)$, which is defined as follows.

Definition 11. Union similarity upper bound *between a bit vector $BV(u_i)$ and a union bit vector $BV_{\cup}(v)$ is:*

$$UB'(BV(u_i), BV_{\cup}(v)) = \frac{\sum_{a\in BV(u_i)\wedge BV_{\cup}(v)} W(a)}{\sum_{a\in BV(u_i)} W(a)}. \tag{5}$$

Based on Definitions 10 and 11, we have the following aggregation dominance principle.

Theorem 3 (Aggregate Dominance Principle). *Given a query signature $Sig(u)$ and a data signature $Sig(v)$, if $UB'(BV(u_i), BV_{\cup}(v)) < \tau$, then for each one-hop neighbor v_j of v, $sim(BV(u_i), BV(v_j)) < \tau$.*

Proof. $sim(BV(u_i), BV(v_j))$

$$= \frac{\sum_{a\in BV(u_i)\wedge BV(v_j)} W(a)}{\sum_{a\in BV(u_i)\vee BV(v_j)} W(a)} \leq \frac{\sum_{a\in BV(u_i)\wedge BV(v_j)} W(a)}{\sum_{a\in BV(u_i)} W(a)}$$
$$\leq \frac{\sum_{a\in BV(u_i)\wedge BV_{\cup}(v)} W(a)}{\sum_{a\in BV(u_i)} W(a)} = UB'(BV(u_i), BV_{\cup}(v)) < \tau.$$

□

Example 7. Considering a one-hop neighbor u_3 of the query vertex u_1 in Fig. 1a, where $BV(u_3) = 0101$, and one-hop neighbors v_2 and v_4 of the data vertex v_5, where $BV(v_2) = 0011$ and $BV(v_4) = 1010$. Since $BV_{\cup}(v) = BV(v_2) \vee BV(v_4) = 1011$, $UB'(BV(u_3), BV_{\cup}(v)) < 0.6 = \tau$. Based on Theorem 3, $sim(BV(u_3), BV(v_j)) < \tau$, where $j = 2, 4$. Therefore, v_5 is not a candidate vertex of u_1 even though $S(u_1) = S(v_5)$.

Based on aggregate dominance principle, we have the following Lemmas of Theorem 3.

Lemma 1. *Given a query signature $Sig(u)$ and a bucket signature $Sig(B)$, assume bucket B contains n data signatures $Sig(v_t)$ $(t = 1, \ldots, n)$, if $UB'(BV(u), \vee BV(v_t)) < \tau$ or there exists at least one neighboring vertex u_i $(i = 1, \ldots, m)$ of u such that $UB'(BV(u_i), \vee BV_{\cup}(v_t)) < \tau$, then all data signatures in bucket B can be pruned.*

Lemma 2. *Given a query signature $Sig(u)$ and a data signature $Sig(v)$, if $sim(BV(u), BV(v)) < \tau$ or there is at least one neighboring vertex u_i $(i = 1, \ldots, m)$ of u such that $UB'(BV(u_i), BV_{\cup}(v)) < \tau$, $Sig(v)$ can be pruned.*

Proofs of Lemmas 1 and 2 are provided in Appendix B, available in the online supplemental material.

In summary, structural pruning works as follows. We first prune the signature buckets that do not contains candidates of query vertices Then, we further prune buckets as a whole based on Lemma 1. For the each candidate v in a bucket B that cannot be pruned, we sequentially check the similarity constraints between $Sig(u)$ and $Sig(v)$, and prune $Sig(v)$ based on Lemma 2. The aggregate dominance principle guarantees that structural pruning will not prune legitimate candidates. Therefore, the results are exact.

6 Dominating-Set-Based Subgraph Matching

In this section, we propose an efficient dominating-set-based subgraph matching algorithm (denoted by *DS-Match*) facilitated by a dominating set selection method.

6.1 DS-Match Algorithm

DS-Match algorithm first finds matches of a *dominating query graph (DQG)* Q^D (defined in Definition 14) formed by the vertices in dominating set $DS(Q)$, then verifies whether each match of Q^D can be extended as a match of Q. DS-Match is motivated by two observations: First, compared with typical subgraph matching over vertex-labeled graph, the overhead of finding candidates in SMS^2 queries is relatively higher, as the computation cost of set similarity is much higher than that of label matching. We can save filtering cost by only finding candidate vertices for dominating vertices rather than all vertices in Q. Second, we can speed up subgraph matching by only finding matches of dominating query vertices. The candidates of remaining (non-dominating) query vertices can be filled up by the structural constraints between dominating vertices and non-dominating vertices. In this way, the size of intermediate results during subgraph matching can be greatly reduced.

In the following, we formally define the dominating set.

Definition 12 (Dominating Set). *Let $Q = (V, E)$ be a undirected, simple graph without loops, where V is the set of*

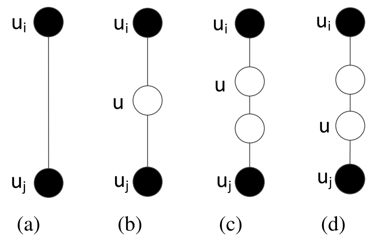


Fig. 6. Possible topologies of neighboring dominating vertices.

vertices and E is the set of edges. A set $DS(Q) \subseteq V$ is called a dominating set *for Q if every vertex of Q is either in $DS(Q)$, or adjacent to some vertex in $DS(Q)$.*

Based on Definition 12, we have the following Theorem.

Theorem 4. *Assume that u is a dominating vertex in Q's dominating set $DS(Q)$. If $|DS(Q)| \geq 2$. Then, there exists at least one vertex $u' \in DS(Q)$ such that $Hop(u, u') \leq 3$, where $Hop(\cdot,\cdot)$ is the minimal number of hops between two vertices. The dominating vertex u' is called a* neighboring dominating vertex *of u.*

Proof. Assume there does not exist any vertex $u' \in DS(Q)$ such that $Hop(u, u') \leq 3$, then there can be at least three non-dominating vertex on the path between u and any other vertex $u' \in DS(Q)$. In this case, at least one non-dominating vertex is not adjacent to u or u', which contradicts with Definition 12. Hence, the theorem holds. □

Definition 13. *Given a vertex u in a graph Q, u's one-hop neighbor set ($N_1(u)$) and two-hop neighbor set ($N_2(u)$) are defined as follows: $N_1(u) = \{u'|$ there exists a length-1 path connecting u and $u'\}$; and $N_2(u) = \{u'|$ there exists a length-2 path connecting u and $u'\}$.*

As shown in Fig. 6, there are four possible topologies of the shortest path between two neighboring dominating vertices u_i and u_j. Based on Theorem 4, we define the *dominating query graph* as follows:

Definition 14 (Dominating Query Graph). *Given a dominating set $DS(Q)$, the dominating query graph Q^D is defined as $\langle V(Q^D), E(Q^D)\rangle$, and there is an edge (u_i, u_j) in $E(Q^D)$ iff at least one of the following conditions holds:*

1) u_i is adjacent to u_j in query graph Q (Fig. 6a);
2) $|N_1(u_i) \cap N_1(u_j)| > 0$ (Fig. 6b);
3) $|N_1(u_i) \cap N_2(u_j)| > 0$ (Fig. 6c);
4) $|N_2(u_i) \cap N_1(u_j)| > 0$ (Fig. 6d).

where the conditions correspond to the four possible topologies in Fig. 6, respectively. In Condition 1), the edge weight (i.e., the path length between u_i and u_j) is 1. In Condition 2), the edge weight is 2. In Conditions 3) and 4), the edge weights are both 3.

To transform a query graph Q to a dominating query graph Q^D, we first find a dominating set $DS(Q)$ of Q. Then for each pair of vertices u_i, u_j in $DS(Q)$, we determine whether there is an edge (u_i, u_j) between them and the weight of (u_i, u_j) according to Definition 14.

Example 8. Fig. 7 shows an example of a dominating query graph Q^D of the query graph Q in Fig. 1a. The dominating set of Q contains u_1 and u_3. Note that, edge (u_1, u_3) in Q^D has two weights "1, 2". The reason is that u_1 is adjacent to u_3 and $|N_1(u_1) \cap N_1(u_3)| > 0$ in Q.

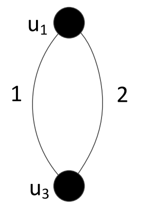


Fig. 7. Example of a dominating query graph.

To find matches of dominating query graph, we propose the distance preservation principle.

Theorem 5 (Distance Preservation Principle). *Given a subgraph match X^D of Q^D in data graph G, Q^D and X^D have n vertices $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$, respectively, where $v_i \in C(u_i)$. Considering an edge (u_i, u_j) in Q^D, then all the following* distance preservation principles *hold:*

1) if the edge weight is 1, then v_i is adjacent to v_j;
2) if the edge weight is 2, then $|N_1(v_i) \cap N_1(v_j)| > 0$;
3) if the edge weight is 3, then $|N_1(v_i) \cap N_2(v_j)| > 0$ or $|N_2(v_i) \cap N_1(v_j)| > 0$.

Proof. Since X^D should preserve the same structural information of Q^D, based on Definition 13, the above distance preservation principles can be proved. □

Now, we propose a dominating query graph match algorithm based on distance preservation principle to find matches of the dominating query graph. A subgraph match is a mapping function M from $V(Q^D)$ to $V(X)$, where X is a matching subgraph of Q^D in data graph G. The process of finding the mapping function can be described by means of *State Space Representation* (SSR) [14]. Each state s is associated with a partial mapping solution $M(s)$, which contains only a subset of M.

Algorithm 1. DQG-Match

```
Input: a dominating query graph $Q^D$, an intermediate state $s$;
       the initial state $s_0$ has $M(s_0) = \phi$
Output: the mapping $M(Q^D)$ between $Q^D$ and $G$'s subgraph
1 if $M(s)$ covers all the vertices of $Q^D$ then
2    $M(Q^D)$=$M(s)$;
3    Output $M(Q^D)$;
4 else
5    Compute the set $PA(s)$ of all the pairs $(u, v)$, where
     $u \in V(Q^D)$ and $v \in V(G)$;
6    for each pair $(u, v)$ in $PA(s)$ do
7       if $(u, v)$ satisfies the distance preservation principle then
8          Add $(u, v)$ to $M(s)$ and compute current state $s'$;
9          Call DQG-Match($s'$);
```

As shown in Algorithm 1, the current state s is initially set to ϕ. We build a candidate pair set $PA(s)$ containing all the possible candidate pairs (u, v) and add it to the current state s (Line 5). When a candidate pair (u, v) is added to the current mapping $M(s)$, we verify if the partial match $M(s)$ satisfies the distance preservation principle (Lines 6-7). If yes, we continue to explore the state until a subgraph match of Q^D is found (Lines 8-9). Otherwise, the corresponding search branch is terminated. For example, given the dominating query graph Q^D in Fig. 7, $\{v_2\}$ and $\{v_3\}$ are matching vertices of u_1 and u_3 in Q^D, respectively. Thus, the mapping function M is $\{(u_1, v_2), (u_3, v_3)\}$.

In the following, we propose DS-match algorithm (Algorithm 2). First, Algorithm 1 is called to find the mapping function $M(Q^D)$ (Line 2). The mapping function $M(s)$ of current state s is initialized to $M(Q^D)$ (Line 3). Second, DS-match extends each match of the dominating query graph Q^D to a subgraph match of the query graph Q. If $M(s)$ covers all the vertices of Q, then we output the mapping function $M(Q)$ (i.e. a subgraph match of Q) (Lines 4-6). Otherwise, for each non-dominating vertex $u' \in V(Q) - V(Q^D)$, we considering one-hop and two-hop neighboring dominating vertices of u' (i.e. $N_1(u')$ and $N_2(u')$) (Lines 7-9). Note that, for each dominating vertex $u_i \in N_1(u')$ and $u_j \in N_2(u')$, candidate vertex sets $C(u_i)$ and $C(u_j)$ have been found by Algorithm 1. Based on distance preservation principle, each candidate vertex v' of u' must be one-hop neighbor and two-hop neighbor of the vertices in $C(u_i)$ and $C(u_j)$, respectively (Line 10). Then, we check whether the candidate pair (u', v') satisfies conditions of subgraph match with set similarity (Definition 1) (Line 11). If yes, (u', v') is added to the current state (Line 12). We continue to explore the state until all non-dominating vertices are considered (Line 13).

Algorithm 2. DS-Match

```
Input: a query graph $Q$, a dominating query graph $Q^D$, and an
       intermediate state $s$; the initial state $s_0$ has $M(s_0) = \phi$
Output: the mapping $M(Q)$ between query graph $Q$ and $G'$s
        subgraph
1  if $M(s_0) = \phi$ then
2    Call Algorithm 1 to find $M(Q^D)$;
3    Initialize $M(s)$ with $M(Q^D)$;
4  if $M(s)$ covers all the vertices of $Q$ then
5    $M(Q) = M(s)$;
6    Output $M(Q)$;
7  else
8    for each $u' \in V(Q) - V(Q^D)$ do
9      for each dominating vertex $u_i \in N_1(u')$ and dominating vertex $u_j \in N_2(u')$ do
10       for $v' \in \bigcup_{v \in C(u_i)} N_1(v) \bigcap \bigcup_{v \in C(u_j)} N_2(v)$ do
11         if pair $(u', v')$ satisfies conditions of subgraph match with set similarity then
12           Add $(u', v')$ to $M(s)$ and compute current state $s'$;
13           Call DS-Match($s'$);
```

Example 9. As discussed in Example 8, the non-dominating vertex of the query graph Q in Fig. 1a is u_2. Thus, the neighboring dominating vertices of u_2 are u_1 and u_3. Since the matching vertices of u_1 and u_3 are v_2 and v_3 respectively, the candidate vertex of u_2 is $N_1(v_2) \bigcap N_1(v_3) = v_1$.

6.2 Dominating Set Selection

A query graph may have multiple dominating sets, leading to different performance of SMS^2 query processing. Motivated by such observation, in this subsection, we propose a dominating set selection algorithm to select a *cost-efficient* dominating set of query graph Q, so that the cost of answering SMS^2 query can be reduced.

Intuitively, a cost-efficient dominating set should contain minimum number of dominating vertices, so that the filtering cost can be minimized. In addition, we should also guarantee the number of candidates for each dominating vertex is minimized to reduce the size of intermediate results during subgraph isomorphism.

To problem of finding a cost-efficient dominating set is actually a Minimum Dominating Set (MDS) problem. MDS problem is equivalent to *Minimum Edge Cover* problem [34], which is NP-hard. As a result, we use a best effort *Branch and Reduce algorithm* [34]. The algorithm recursively select a vertex u with *minimum number of candidate vertices* from query vertices that have not been selected, and add to the edge cover an arbitrary edge incident to u. The overhead of finding the dominating set is low, because the query graph is usually small.

Since we do not know a query vertex u's number of candidate vertices before the SMS^2 query is processed, We use a *Hash Sampling* method [35] to make quick estimates of the number. The hash sampling method can construct the sample union $\widetilde{P}_\cup = \widetilde{P}_1 \cup \ldots \cup \widetilde{P}_n$, where $\widetilde{P}_1, \ldots,$ and $\widetilde{P}_n$ are precomputed samples of inverted lists $L(P_1), \ldots,$ and $L(P_n)$. The estimate of the number of u's candidate vertices from the sample can be calculated as

$$A(u) = |A(u)_{\widetilde{P}_\cup}| \frac{|L(P)_\cup|_d}{|\widetilde{P}_\cup|_d}, \tag{6}$$

where $|A(u)_{\widetilde{P}_\cup}|$ is the number of similarity search answers of $S(u)$ on the uniform random sample, $|L(P)_\cup|_d$ is the distinct number of vertex ids contained in the multi-set union $L(P)_\cup$, and $|\widetilde{P}_\cup|_d$ is the distinct number of vertex ids in the sample union.

Example 10. In Fig. 1, we find u_3 has the minimum number of candidates, i.e., v_3. Thus, $\{u_1, u_3\}$ is selected as the cost-efficient dominating set according to the branch and reduce algorithm.

7 Experimental Evaluation

7.1 Datasets and Setup

All experiments are implemented in C++, and conducted on a 2.5 GHZ Core class machine with 4 GB RAM. The operating system is Windows 7 Ultimate edition. We use two real datasets Freebase and the DBpedia, and synthetic graphs that follow scale-free graph model. The datasets used in our experiment are described as follows.

1) Freebase (http://www.freebase.com) is a large collaborative knowledge base of structured data. We use the entity relation graph of Freebase (denoted by FB), in which each vertex represents an entity, such as an actor, a movie, etc., and each edge represents the relationship between two entities. Each vertex is associated with a set of elements, which describes features of the corresponding entity. The weight of each feature specifies its importance, which is

normalized to the range of $[0, 1]$. This graph contains 1,047,829 nodes and 18,245,370 edges. The number of distinct elements is 243, and the average number of elements of each vertex is 6.

2) DBpedia (http://dbpedia.org) extracts structured information from Wikipedia (http://www.wikipedia.org). In our DBpedia dataset (denoted by DBP), each vertex corresponds to an entity (i.e., article) extracted from Wikipedia, which contains a set of words (tokens). The weight of each word is specified by TF/IDF. We use the classical feature selection method [36] to select 2,000 words with the highest TF/IDF value in the DBPedia as the elements. This graph contains 1,010,205 nodes and 1,588,181 edges. The average number of elements of each vertex is 20.6.
3) We generate synthetic scale-free data graphs (denoted by SF) using the graph generator of [37], in which node degree follows power law distribution. Three SF datasets are used in our experiment: SF1M, SF5M and SF10M which contain 1 million, 5 million and 10 million vertices, 1,260,704, 6,369,405 and 24,750,113 edges, respectively. The number of distinct elements is 100, and the average number of elements of each vertex is 5.5. Each element is randomly assigned a weight in the range of $[0, 1]$. The weight distribution among all the elements follows Uniform distribution by default. SF1M is the default SF dataset.

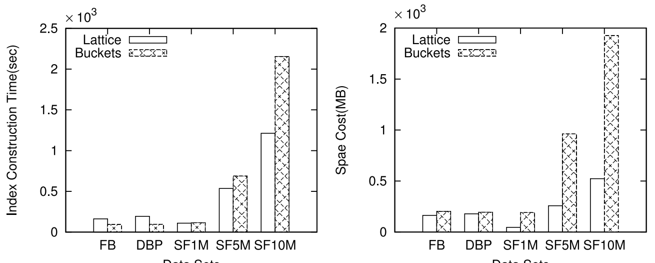
According to a recent experimental track paper [38], the graph with 10 million vertices and 24 million edges used in our experiment is the largest data graph reported in a *single* machine in the literatures on subgraph search. Although there is a recent work STW on subgraph search over a billion node graph, it works on a cloud of eight machines with 32 GB RAM and two four-core CPUs [10].

We extract 100 query graphs Q from data graph G by starting from a random vertex in G and then traversing the graph through connecting random edges, where the maximum number, n_{max}, of vertices in query graph Q is set to {3, **5**, 8, 10, 12}. The similarity threshold τ is set to {0.5, 0.6, 0.7, 0.8, **0.9**}, and the support threshold $minsup$ is set to **30,000**. The numbers in bold font are default values. In subsequent experiments, each time we vary one parameter, other parameters are set to default values. The performance of SMS^2 queries is measured in terms of the query response time per query graph, and the average number of candidates of each query vertex. The sizes of FB, DBP, SF1M, SF5M and SF10M are 118, 51, 38, 206 and 415 MB, respectively.

7.2 Competing Methods

We compare our method (denoted by SMS^2) with the baseline method (denoted by BL). BL method builds an inverted index for all the element sets of data vertices, based on which it searches each query vertex u's candidates. After candidate search, BL finds matching subgraphs of the query graph using QuickSI algorithm [19].

We also compare SMS^2 with existing exact subgraph matching methods including Turbo$_{ISO}$[15], GADDI [18], SPath [8], GraphQL [21], STW [10], D-Join[9] and R-join [2]. Since STW is a distributed graph matching algorithm, for fair comparison, we implement STW using the same method as [15] in *one* machine. These methods first find candidate subgraphs that are isomorphic to the query graph Q, and then find the subgraph matches by checking set similarity of each pair of matching vertices.



(a) Index Construction Time (sec) (b) Space Cost

Fig. 8. Offline performance vs. datasets, $minsup = 30{,}000$.

To evaluate the performance of set similarity pruning, structural pruning and DS-match algorithms, we compare our method with SMS^2-S, SMS^2-Q and SMS^2-R, respectively. SMS^2-S method only uses set similarity pruning to find candidate vertices of each dominating query vertex. Then, it employs DS-match algorithm to find matching subgraphs. SMS^2-Q method finds candidate vertices of all query vertices instead of dominating query vertices using the proposed pruning techniques including set similarity pruning and structure-based pruning. Then, it employs the QuickSI algorithm [19] to find matching subgraphs based on candidates. The only difference between SMS^2-R and SMS^2 is that SMS^2-R randomly chooses a dominating set of the query graph, while SMS^2 uses dominating set selection algorithm to select a cost-efficient dominating set.

7.3 Offline Performance

Index Construction Time. Fig. 8a reports the time cost of index construction on real/synthetic datasets. For the inverted pattern lattice (denoted by Lattice), the construction time ranges from 109.6 seconds to 1,212.3 seconds. For signature buckets (denoted by Buckets), the construction time ranges from 115 seconds to 2,154.2 seconds.

Space cost. The space cost of inverted pattern lattice includes the space of the lattice in the main memory and the inverted lists in the disk. The space cost of signature buckets is the size of signature buckets of all the data vertices. As analyzed in Appendix D, available in the online supplemental material, the space complexity of the inverted pattern lattice and signature buckets are $O(2^{|\mathcal{U}|})$ and $O(|V(G)|)$ respectively, where $|\mathcal{U}|$ is the number of distinct elements in element universe $\mathcal{U}$. As shown in Fig. 8b, the space cost of inverted pattern lattice ranges from 45.9 to 523 MB, and the space cost of signature buckets ranges from 192 to 1,926.4 MB. We can find that the space cost is proportional to the size of each dataset.

7.4 Online Performance

7.4.1 Performance vs. Datasets

In this subsection, we first compare SMS^2 with the competing methods on an unlabeled and single-labeled data graph, respectively. Then, we compare SMS^2 with the BL method.

TABLE 1
Query Response Time (sec) on Unlabeled Graph

Dataset	SMS^2	$Turbo_{ISO}$
FB	35.7	Fail
DBP	33.2	5829.5
SF1M	43.8	6237.1
SF5M	170.2	Fail
SF10M	370.5	Fail

Since set similarity becomes invalid for an unlabeled or single-labeled graph, we turn off the set similarity pruning process in SMS^2. In addition, we use hashing mechanism rather than union similarity upper bound (in structural pruning) to directly locate candidate signature buckets.

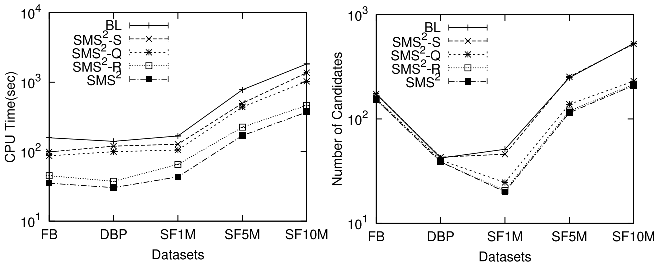
To enable the comparison on an unlabeled graph, we ignore all sets associated with each vertex and apply competing methods to find subgraph matches. Unfortunately, all competitors except $Turbo_{ISO}$ fail to finish their query processing in a reasonable time. The reason is that unlabeled vertices generate a large number of intermediate results. Although $Turbo_{ISO}$ is the only competitor survived in DBP and SF1M, the expensive verification process leads to long query response time. As shown in Table 1, our method outperforms $Turob_{ISO}$ by up to 175.6 times.

For a single-labeled graph, we assign a single label to each vertex and use multiple distinct labels on all vertices. The distinct number of labels is set to 10 percent of the total number of vertices. The similarity threshold τ is set to 1. As a result, the SMS^2 query is degraded to exact subgraph search. We compare the query response time of SMS^2 with the state-of-the-art subgraph isomorphism algorithm $Turbo_{ISO}$ [15]. $Turbo_{ISO}$ also fails to finish its query processing in SF5M and SF10M datasets, respectively. Thus, we use SF1.5M and SF2M datasets instead. As shown in Table 2, SMS^2 results in shorter query response time than $Turbo_{ISO}$, which proves that the proposed structural pruning and DS-match algorithms can be efficiently applied to exact subgraph search. Note that, the inverted pattern lattice for a single-labeled graph is actually an inverted index consists of a list of all the distinct labels, and for each label, a list of vertices in which it belongs to.

As shown in Fig. 9a, SMS^2 results in much shorter query response time than BL, SMS^2-S, SMS^2-Q, and SMS^2-R on both real and synthetic datasets. The query response time of SMS^2 changes from 35.17 seconds to 370.49 seconds. SMS^2 outperforms SMS^2-S because SMS^2 uses both set similarity pruning and structure-based pruning while SMS^2-S only uses set similarity pruning. SMS^2 outperforms SMS^2-Q by at least 58 percent query response time. This is because

TABLE 2
Query Response Time (sec) on Exact Subgraph Search

Dataset	SMS^2	$Turbo_{ISO}$
FB	1.55	6.43
DBP	0.67	0.75
SF1M	1.14	1.41
SF1.5M	1.54	3.01
SF2M	2.03	6.68



(a) Query Response Time (sec) (b) Number of Candidates

Fig. 9. Performance vs. datasets, $n_{max} = 5$, $\tau = 0.9$.

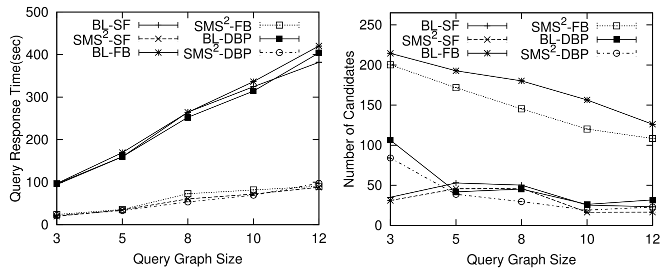
SMS^2 saves pruning cost by only finding candidates of dominating query vertices. In addition, DS-match algorithm has better performance than QuickSI because it saves the subgraph matching cost by reducing the number of intermediate results. Since SMS^2 employs dominating set selection algorithm to select a cost-efficient dominating query graph, SMS^2 has better performance than SMS^2-R which randomly selects the dominating set. SMS^2 outperforms BL because set similarity pruning and structure-based pruning techniques result in less candidates and pruning cost compared with the inverted-index-based similarity search, and DS-match algorithm has better performance than QuickSI algorithm. As analyzed in Appendix C, available in the online supplemental material, the time complexities of set similarity pruning and structure-based pruning techniques are $O(|I|)$ and $O(|\bigcup_{u \in DS(Q)} C(u)|)$ respectively. We can also observe from Fig. 9a that the query response time of SMS^2 grows linearly with the size of data graph from 1 million to 10 million, which indicates the scalability of our method.

As shown in Fig. 9b, SMS^2-Q, SMS^2-R and SMS^2 generate similar number of candidates, because these method use the same pruning techniques. SMS^2-S and BL also generate similar number of candidates, because they both consider weighted set similarity between each query vertex and data vertex. SMS^2 generates smaller number of candidates than BL by at least 8.5 percent, and at most 60 percent on different datasets. These results indicate that structure-based pruning technique can prune at least 8.5 percent candidates, and set similarity pruning technique can prune at least 40 percent candidates. It is worth noting that the number of candidates shows a similar growth trend to the size of data graph. For example, there are 43.4, 170 and 370.5 candidates for datasets SF1M, SF5M and SF10M.

7.4.2 Performance vs. Query Graph Size

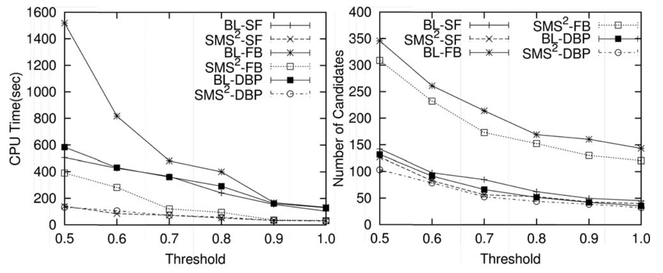
In this subsection, we compare SMS^2 with BL by varying query graph size (i.e., number of query vertices in query graph) from 3 to 12. To represent different datasets, BL and SMS^2 are further divided into BL-SF, SMS^2-SF, BL-FB, SMS^2-FB, BL-DBP, SMS^2-DBP.

As shown in Figs. 10a and 10b, the query response time of SMS^2 increases much slower than that of BL as the query graph size changes from 3 to 12. This is because BL incurs much more overhead than SMS^2 in both pruning phase and subgraph matching phase. The above results also confirm that SMS^2 are more scalable than BL against different query graph sizes. From Fig. 10b, the number of candidates of SMS^2 and BL decreases as query graph size increases, and



(a) Query Response Time (sec) (b) Number of Candidates

Fig. 10. Performance vs. query graph size, $\tau = 0.9$.



(a) Query Response Time (sec) (b) Number of Candidates

Fig. 11. Performance vs. Similarity Threshold, $n_{max} = 5$.

SMS^2 results in smaller number of candidates than BL. This is because a small query graph probably has more subgraph matches than a large query graph, and the pruning techniques of SMS^2 have greater pruning power than that of BL. Note that, although SMS^2-FB generates more candidates than BL-SF and BL-DBP, it results in shorter query response time. The reason is that both set similarity pruning and structure-based pruning save much query processing cost compare to existing methods.

7.4.3 Performance vs. Properties of Element Sets

The performance of the set similarity pruning techniques highly depends on the element sets of query vertices. In this subsection, we evaluate how the query response time and the average number of candidates of each vertex are affected by the following types of sets. These sets contain: high *term frequency* elements (the term frequency of all elements is larger than 0.98, denoted by HighTF), and low term frequency elements (the term frequency of all elements is lower than 0.01, denoted by LowTF), large number of elements (the number of elements is larger than 80, denoted by LargeN), small number of elements (the number of elements is smaller than 5, denoted by SmallN), respectively. We generate query graphs that only contain the vertices that have one of the four element set types.

From Table 3, we observe that the query performance degrades as the term frequency of elements become higher. This is because there are more candidates of a query vertex with HighTF set than that of a query vertex with LowTF set. Moreover, the sizes of inverted lists of high term frequency elements are larger than that of low term frequency elements, which also contributes to higher query response time of query vertices with HighTF sets.

Table 3 demonstrates that queries containing LargeN sets usually have better performance than queries with SmallN sets. The reason is that LargeN leads to smaller number of candidates than SmallN. For LargeN, the pruning power of vertical pruning decreases as longer prefix (larger p) will be found, while the pruning power of horizontal pruning increases as more small size frequent patterns will be pruned. For SmallN, the pruning power of both vertical pruning and horizontal pruning increases as shorter prefix will be left and larger size frequent patterns will be pruned.

TABLE 3
Impact of Properties of Element Sets, $n_{max} = 5$, $\tau = 0.9$

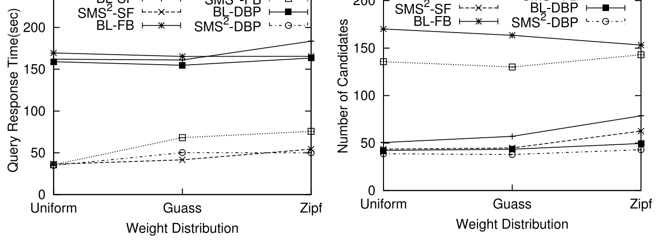
		HighTF	LowTF	LargeN	SmallN
BL-FB	Time (sec)	205.7	120.99	132.60	175.09
	Candidates	60.56	23.54	16.30	42.51
SMS^2-FB	Time (sec)	24.62	4.52	7.98	19.78
	Candidates	51.82	1.78	0.65	25.09
BL-DBP	Time (sec)	196.84	145.53	70.27	188.56
	Candidates	52.51	37.27	12.20	78.56
SMS^2-DBP	Time (sec)	28.88	7.51	20.13	42.06
	Candidates	8.27	5.35	0.88	60.67
BL-SF	Time (sec)	230.65	180.41	107.52	259.23
	Candidates	56.52	51.21	8.70	64.20
SMS^2-SF	Time (sec)	35.01	34.11	9.22	44.34
	Candidates	4.53	4.42	0.76	56.70

7.4.4 Performance vs. Similarity Threshold

We compare BL with SMS^2 by varying the similarity threshold τ from 0.5 to 1 in different datasets. As shown in Figs. 11a and 11b, SMS^2 has much shorter query response time and smaller number of candidates than BL, especially when the threshold is small. For example, when $\tau = 0.5$, the gap between the average number of candidates of SMS^2-FB and BL-FB is less than 37, while SMS^2-FB reduces more than 1,130 seconds query response time compared to BL-FB. This is because larger similarity threshold will result in fewer candidate vertices, thus reducing the query response time. Note that, when τ is set to 1, the problem is degraded to an exact subgraph search problem. In such case, the set similarity pruning is still valid.

7.4.5 Performance vs. Weight Distribution

Finally, we compare SMS^2 with BL with different weight distributions of the elements: Uniform distribution, Gaussian distribution, and Zipf distribution. We observe from Fig. 12 that the performance does not change much as the weight



(a) Query Response Time (sec) (b) Number of Candidates

Fig. 12. Performance vs. Wt. distribution, $n_{max} = 5$, $\tau = 0.9$.

distribution varies. This is expected, because the similarity function and upper bounds cannot be strongly affected by the weight distribution. The results in Fig. 12 confirm that our approach greatly outperforms BL.

8 Conclusions

In this paper, we study the problem of subgraph matching with set similarity, which exists in a wide range of applications. To tackle this problem, we propose efficient pruning techniques by considering both vertex set similarity and graph topology. A novel inverted pattern lattice and structural signature buckets are designed to facilitate the online pruning. Finally, we propose an efficient dominating-set-based subgraph match algorithm to find subgraph matches. Extensive experiments have been conducted to demonstrate the efficiency and effectiveness of our approaches.

Acknowledgments

The corresponding author of this paper is Lei Zou. Liang Hong and Lei Zou are supported by National Science Foundation of China (NSFC) under Grant 61303025, 61370055, and Tencent. Philip S. Yu is supported by NSF through grants CNS-1115234, and OISE-1129076, US Department of Army through grant W911NF-12-1-0066 and the Pinnacle Lab at Singapore Management University.

References

[1] B. Cui, H. Mei, and B. C. Ooi, “Big data: The driver for innovation in databases,” *Nat. Sci. Rev.*, vol. 1, no. 1, pp. 27–30, 2014.

[2] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang, “Fast graph pattern matching,” in *Proc. Int. Conf. Data Eng.*, 2008, pp. 913–922.

[3] X. Zhu, S. Song, X. Lian, J. Wang, and L. Zou, “Matching heterogeneous event data,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1211–1222.

[4] S. Bruckner, F. Huffner, R. M. Karp, R. Shamir, and R. Sharan, “Torque: Topology-free querying of protein interaction networks,” *Nucleic Acids Res.*, vol. 37, no. suppl 2, pp. W106–W108, 2009.

[5] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, “Parallel subgraph listing in a large-scale graph,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 625–636.

[6] Y. Shao, L. Chen, and B. Cui, “Efficient cohesive subgraphs detection in parallel,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 613–624.

[7] Y. Tian and J. M. Patel, “Tale: A tool for approximate large graph matching,” in *Proc. 24th Int. Conf. Data Eng.*, 2008, pp. 963–972.

[8] P. Zhao and J. Han, “On graph query optimization in large networks,” *Proc. VLDB Endowment*, vol. 3, nos. 1/2, pp. 340–351, 2010.

[9] L. Zou, L. Chen, and M. T. Ozsu, “Distance-join: Pattern match query in a large graph database,” *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 886–897, 2009.

[10] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, “Efficient subgraph matching on billion node graphs,” *Proc. VLDB Endowment*, vol. 5, no. 9, pp. 788–799, 2012.

[11] W. Lu, J. Janssen, E. Milios, N. Japkowicz, and Y. Zhang, “Node similarity in the citation graph,” *Knowl. Inf. Syst.*, vol. 11, no. 1, pp. 105–129, 2006.

[12] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, and Z. Ives, “Dbpedia: A nucleus for a web of open data,” in *Proc. 6th Int. Semantic Web*, 2007, pp. 722–735.

[13] M. Hadjieleftheriou and D. Srivastava, “Weighted set-based string similarity,” *IEEE Data Eng. Bull.*, vol. 33, no. 1, pp. 25–36, Mar. 2010.

[14] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, “A (sub)graph isomorphism algorithm for matching large graphs,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, Oct. 2004.

[15] W.-S. Han, J. Lee, and J.-H. Lee, “Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 337–348.

[16] H. He and A. K. Singh, “Closure-tree: An index structure for graph queries,” in *Proc. 222nd Int. Conf. Data Eng.*, 2006, p. 38.

[17] J. R. Ullmann, “An algorithm for subgraph isomorphism,” *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[18] S. Zhang, S. Li, and J. Yang, “Gaddi: Distance index based subgraph matching in biological networks,” in *Proc. 12th Int. Conf. Extending Database Technol.: Adv. Database Technol.*, 2009, pp. 192–203.

[19] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, “Taming verification hardness: An efficient algorithm for testing subgraph isomorphism,” *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 364–375, 2008.

[20] R. Di Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, and D. Shasha, “Sing: Subgraph search in non-homogeneous graphs,” *BMC Bioinformat.*, vol. 11, no. 1, p. 96, 2010.

[21] H. He and A. K. Singh, “Graphs-at-a-time: Query language and access methods for graph databases,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 405–418.

[22] G. Gülsoy and T. Kahveci, “Rinq: Reference-based indexing for network queries,” *Bioinformatics*, vol. 27, no. 13, pp. 149–158, 2011.

[23] V. Memišević and N. Pržulj, “C-graal: Common-neighbors-based global graph alignment of biological networks,” *Integr. Biol.*, vol. 4, no. 7, pp. 734–743, 2012.

[24] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel, “Saga: A subgraph matching tool for biological graphs,” *Bioinformatics*, vol. 23, no. 2, pp. 232–239, 2007.

[25] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, “Capturing topology in graph pattern matching,” *Proc. VLDB Endowments*, vol. 5, no. 4, pp. 310–321, 2012.

[26] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan, “Nema: Fast graph search with label similarity,” in *Proc. VLDB Endowment*, vol. 6, no. 3, pp. 181–192, 2013.

[27] X. Yu, Y. Sun, P. Zhao, and J. Han, “Query-driven discovery of semantically similar substructures in heterogeneous networks,” in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2012, pp. 1500–1503.

[28] L. Zou, L. Chen, and Y. Lu, “Top-k subgraph matching query in a large graph,” in *Proc. ACM 1st PhD Workshop CIKM*, 2007, pp. 139–146.

[29] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava, “Fast indexes and algorithms for set similarity selection queries,” in *Proc. 24th Int. Conf. Data Eng.*, 2008, pp. 267–276.

[30] R. J. Bayardo, Y. Ma, and R. Srikant, “Scaling up all pairs similarity search,” in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 131–140.

[31] D. Xin, J. Han, X. Yan, and H. Cheng, “Mining compressed frequent-pattern sets,” in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 709–720.

[32] S. Chaudhuri, V. Ganti, and R. Kaushik, “A primitive operator for similarity joins in data cleaning,” in *Proc. 22nd Int. Conf. Data Eng.*, 2006, p. 5.

[33] A. Gionis, P. Indyky, and R. Motwaniz, “Similarity search in high dimensions via hashing,” in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 518–529.

[34] F. V. Fomin, F. Grandoni, and D. Kratsch, “A measure and conquer approach for the analysis of exact algorithms,” *J. ACM*, vol. 56, no. 5, p. 25, 2009.

[35] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava, “Hashed samples: Selectivity estimators for set similarity selection queries,” in *Proc. 25th Int. Conf. Very Large Data Bases*, 2008, pp. 518–529.

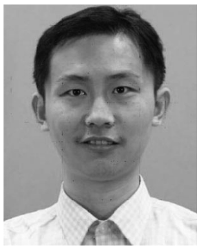
[36] Y. Yang and P. J. O., “A comparative study on feature selection in text categorization,” in *Proc. 14th Int. Conf. Mach. Learn.*, 1997, pp. 412–420.

[37] F. Viger and M. Latapy, “Efficient and simple generation of random simple connected graphs with prescribed degree sequence,” in *Proc. 11th Annu. Int. Conf. Comput. Combinatoric*, 2005, pp. 440–449.

[38] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, “An in-depth comparison of subgraph isomorphism algorithms in graph databases,” *Proc. VLDB Endowment*, vol. 6, no. 2, pp. 133–144, 2012.

Liang Hong received the BS and PhD degrees in computer science from the Huazhong University of Science and Technology (HUST) in 2003 and 2009, respectively. Now, he is an associate professor in the School of Information Management at Wuhan University. His research interests include graph database, spatio-temporal data management, and social networks. He is member of the IEEE.

Lei Zou received the BS degree and PhD degree in computer science from the Huazhong University of Science and Technology (HUST) in 2003 and 2009, respectively. He is currently an associate professor in the Institute of Computer Science and Technology of Peking University. His research interests include graph database and semantic data management. He is member of the IEEE.



Xiang Lian received the BS degree from the Department of Computer Science and Technology, Nanjing University, in 2003, and the PhD degree in computer science from the Hong Kong University of Science and Technology, Hong Kong. He is currently an assistant professor in the Department of Computer Science at the University of Texas-Pan American. His research interests include probabilistic/uncertain data management and probabilistic RDF graphs. He is member of the IEEE.

Philip S. Yu is a distinguished professor of computer science at the University of Illinois at Chicago and also holds the Wexler chair in information technology. He was a manager of the Software Tools and Techniques Group at the Watson Research Center of IBM. His research interest is on big data, including data mining, data stream, database and privacy. He has published more than 860 papers in refereed journals and conferences. He holds or has applied for more than 250 US patents. He is a fellow of the ACM and the IEEE.