Diversified Top-k Subgraph Querying in a Large Graph

Zhengwei Yang Ada Wai-Chee Fu Ruifeng Liu
The Chinese University of Hong Kong
{zwyang,adafu,rfliu}@cse.cuhk.edu.hk

ABSTRACT

Subgraph querying in a large data graph is interesting for different applications. A recent study shows that top-k diversified results are useful since the number of matching subgraphs can be very large. In this work, we study the problem of top-k diversified subgraph querying that asks for a set of up to k subgraphs isomorphic to a given query graph, and that covers the largest number of vertices. We propose a novel level-based algorithm for this problem which supports early termination and has a theoretical approximation guarantee. From experiments, most of our results on real datasets used in previous works are near optimal with a query time within 10ms on a commodity machine.

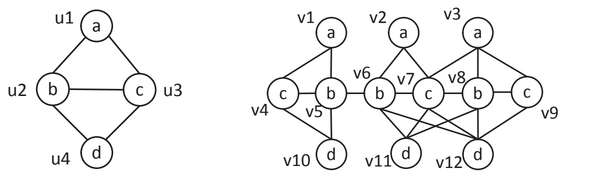
Keywords

subgraph isomorphism; top-k; diversity; maximum k-coverage

1. INTRODUCTION

Graph databases have been found useful in social networks, RDF, and many other applications. An important processing for graph data applications is that of the subgraph isomorphism search. Given a data graph and a query graph, the querying returns subgraphs in the data graph that are isomorphic to the query graph. Though this is an NP-hard problem, it has been found to be solvable with good response time for many real data graphs [6, 16, 26, 35, 21, 15]. The problem is manageable since the query graph is quite small in most applications. However, as data graphs are growing in size, the number of isomorphic subgraphs in such a graph can be excessively large. Most existing works stop their computation when a fixed number of subgraphs are found. Unfortunately, the resulting subgraphs are often highly overlapping and not very representative, whilst more interesting solutions may be missed. Such an observation has been made in [10]. They advocate the importance of diversity in the search results. Let us illustrate with an example based on the motivating application in [10].

EXAMPLE 1. A fraction of a collaboration network is shown as G' in Figure 1(b). In this graph, each node represents a person, with a label for the job of the person. The label "a" stands for project manager (PM), "b" for programmer (PRG), "c" for database developer (DB), and "d" for software tester (ST). A company issues a graph query to find a team consisting of a PM, a PRG, a DB, and an ST, such that there is a link between the PM and each of the PRG and DB, the PRG is linked to the DB, and both the PRG and DB are linked to the ST. These requirements are given by the query graph Q in Figure 1(a). There are many matching subgraphs in G'. We limit the number of resulting matchings to a small number k. In this example, let us set $k = 2$. The question is which set of two matchings is better. If we return $(v3, v8, v7, v12)$ and $(v3, v8, v9, v12)$ as matchings for $(u1, u2, u3, u4)$, we have the same PM, PRG and ST in both answers. This limits the choices for the company, since if any of the PM, PRG or ST is not available or not suitable, both results will not be useful. Instead, the disjoint matchings of $(v1, v5, v4, v10)$ and $(v2, v6, v7, v12)$ form a better solution set. Hence, in selecting the top k matchings we aim to reduce the overlapping information among the matchings. That is, the matchings should be diversified.



(a) query graph Q (b) part of the data graph, G'

Figure 1: Diversified subgraph matching; the meanings of the vertex labels are: a - project manager; b - programmer; c - database developer; d - software tester.

The above example shows that top-k diversified subgraph querying (DSQ) can rectify the issues caused by an excessive number of matchings in existing subgraph querying algorithms. Diversity is measured by the number of vertices covered by all the subgraphs in the result. The problem is to find at most k isomorphic subgraphs with a maximum coverage. Diversity has been proposed in [10], but their problem is to search for vertices that match with a single query node, say $u1$ in our example, and they consider graph simulation instead of isomorphism. In Figure 1, for example, [10] considers that $(v2, v6, v7, v11, v12)$ is also a matching to the query graph. To our knowledge, this is the first study to consider DSQ. However, to solve for DSQ, there are some major challenges:
(1) As the number of isomorphic subgraphs is potentially exponential, existing algorithms typically stop after generating around a thousand matchings. With DSQ, we may need to generate many more matchings for selecting the diversified results.
(2) Assuming that we can generate and store all isomorphic sub-

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915216

graphs for a query, the remaining problem then becomes maximum k-coverage, which is also NP-hard. Though a simple greedy algorithm provides an approximation ratio guarantee of over 0.63, it requires scanning all the isomorphic subgraphs k times. However, generating all matchings may already be prohibitively costly.

(3) From the above discussion, a scalable algorithm would need an early termination mechanism so as to return k matchings without an exhaustive search, while having a good approximation guarantee.

Contributions. We study the above challenges and propose a solution for DSQ. Our contributions are summarized as follows.

(1) We propose to study the diversified subgraph querying problem where we ask for top-k diversified isomorphic subgraphs.

(2) We propose a novel solution, called DSQL, for DSQ. DSQL is based on a level-wise subgraph search, where **level** refers to the overlapping size of a newly selected subgraph with a set of collected subgraphs. DSQL has the following desirable properties: (a) an approximation ratio guarantee of $0.25(1 + \max(\frac{1}{k}, \frac{1}{q}))$, where q is the number of vertices in the query graph; (b) supports early termination when k results are obtained.

(3) We propose optimization strategies which greatly improve the performance of DSQL. Two of the strategies can also be used for the subgraph querying problem.

(4) We empirically verify the efficiency and effectiveness of our algorithm. Our experiments show that DSQL often generates near optimal solutions with a fast response time.

(5) Our study leads to some new results for the Maximum k-coverage problem. We propose a multi-scanning technique for a better approximation guarantee, which is asymptotically 0.5. We improve the approximation guarantee of existing streaming algorithms to $0.25(1 + \max(\frac{1}{k}, \frac{1}{q}))$, where q is the number of vertices in the query graph.

The rest of this paper is organized as follows. Section 2 states the problem definition and describes existing methods for related problems. Section 3 gives an overview of our two-phase solution DSQL. Section 4 describes the first phase of DSQL. Section 5 is about the optimization of DSQL. Section 6 is about the second phase of DSQL. The experimental results are given in Section 7. Related works are summarized in Section 8. We conclude in Section 9.

2. PROBLEM DEFINITION AND EXISTING WORKS

Existing work on subgraph querying (SQ) motivates our study. Some preliminary definitions concerning SQ are given below.

Data Graph. We define a data graph as an undirected, vertex-labeled graph $G = (V, E, \Sigma, L)$, where
(1) V is the set of data vertices;
(2) $E \subseteq V \times V$ is a set of undirected edges;
(3) Σ is a set of vertex labels;
(4) for each vertex $v \in V$, $L(v) \in \Sigma$ is the label of v.

Subgraph. A graph $G_s = (V_s, E_s, \Sigma_s, L_s)$ is a *subgraph* of G if $V_s \subseteq V$, $E_s \subseteq E$, $\Sigma_s \subseteq \Sigma$, for each edge $e = (v, v') \in E_s$, $v \in V_s$ and $v' \in V_s$, and for each $v \in V_s$, $L_s(v) = L(v)$.

Query Graph. A query graph is an undirected, vertex-labeled graph $Q = (V_Q, E_Q, \Sigma_Q, L_Q)$, where (1) V_Q, E_Q, and Σ_Q are the sets of query vertices, edges, and vertex labels, respectively; (2) for each vertex $u \in V_Q$, $L_Q(u) \in \Sigma_Q$ is the label of u. Let $q = |Q| = |V_Q|$.

Subgraph Isomorphism (or embedding). Given a data graph $G = (V, E, \Sigma, L)$ and a query graph $Q = (V_Q, E_Q, \Sigma_Q, L_Q)$, a *subgraph isomorphism* is an injective function $f : V_Q \rightarrow V$ such that (1) $L_Q(u) = L(f(u))$ for any vertex $u \in V_Q$; (2) for each edge $(u_i, u_j) \in E_Q$, there exists an edge $(f(u_i), f(u_j)) \in E$.

If an embedding f exists, let G' be the subgraph in G consisting of the vertices $f(u)$ for $u \in V_Q$, and the edges $(f(u_i), f(u_j))$ in (2) above, then we say that G' **is isomorphic to** Q and vice versa. We also say that G' is the subgraph for the embedding f, and that G' is an isomorphic subgraph.

DEFINITION 1 (SUBGRAPH QUERYING -**SQ**). *Given a data graph G and a query graph Q, the subgraph querying problem (SQ) is to find all embeddings of Q in G.*

DEFINITION 2 (COVERAGE). *Given a set $S = \{s_1, ..., s_i\}$ of subgraphs of G, where $s_i = (V_i, E_i)$, the cover set of S, denoted by $C(S)$, is the vertex set $V_C = V_1 \cup V_2 \cup ... \cup V_i$. The coverage of S is given by $|C(S)|$.*

DEFINITION 3 (DIVERSIFIED SUBGRAPH QUERYING - **DSQ**). *Given a data graph G, a query graph Q, and an integer k, the diversified subgraph querying problem (DSQ) is to select a set S of no more than k subgraphs in G that are isomorphic to Q, and such that S has the largest coverage among all such selections of k or less subgraphs.*

Given a data graph G, a query Q and an integer k, diversified subgraph querying returns k subgraphs that are isomorphic to Q and that cover the largest number of vertices possible in G. Unfortunately this problem is NP-hard, a proof of which is given in the appendix.

THEOREM 1. *The problem of DSQ is NP-hard.*

Notice that two embeddings may select the same vertex sets from the data graph G, with different matchings to the query graph. Duplicated vertex sets will not increase the coverage, and thus need not appear in a solution for DSQ. Hence, we are only interested in a set of embeddings with distinct vertex sets.

We overload the term **embedding** to also refer to the vertex set of the subgraph for an embedding isomorphic to Q. Given a set of embeddings, $S = \{s_1, s_2, ..., s_m\}$, for some integer m, denote the set of vertices $s_1 \cup s_2 \cup ... \cup s_m$ as $V(S)$. $C(S) = V(S)$ is the cover set of S. We say that the coverage of S is $|C(S)| = |V(S)|$. For better clarity, we refer to vertices in the query Q as **nodes** and those in the data graph G as **vertices**.

2.1 Disjoint Embeddings

From the definition of DSQ, the best possible coverage for k subgraphs is attained when the subgraphs are disjoint. Therefore, it is interesting to consider an algorithm that returns a maximum number of disjoint isomorphic subgraphs. If the number of returned subgraphs is k or more, we have an optimal solution for DSQ. This problem is also NP-hard, since Maximum 3-dimensional matching is a special case of the problem.

THEOREM 2. *The problem of maximum disjoint isomorphic subgraphs is NP-hard.*

However, a maximum disjoint set may not be necessary for our problem. Instead, a maximal disjoint set of isomorphic subgraphs is already an optimal solution, once there are at least k such subgraphs. This observation is key in our proposed solution, though we generalize it to optimistically look for subgraphs with as little overlapping as possible, progressively relaxing the restriction on the overlapping size.

Algorithm 1: EXISTING SQ FRAMEWORK: QSearch(S,...)

```
1 if $|S| = |V_Q|$ then return $S$ ;
2 $u \leftarrow nextQueryNode(...)$;
3 $cand(u) \leftarrow refineCandidates(u, cand(u), ...)$;
4 foreach $v \in cand(u)$ and $v$ that is not matched do
5     if $IsJoinable(S, u, v)$ then
6         $Update(S, u, v)$; $QSearch(S, ...)$; $Restore(S, u, v)$;
```

2.2 Existing Subgraph Querying Algorithms

DSQ can be solved by first generating all embeddings and then selecting k embeddings with the best coverage. In such a way, DSQ is related to the problems of subgraph querying, SQ, and maximum k-coverage. Here we consider known solutions for SQ. In known applications, query graphs are very small, and a very large number of matching subgraphs are typically obtained from a large data graph. In known existing works for SQ, the search is terminated when k embeddings are found, where k is 1000 in [16, 35] and 1024 in [27, 15]. We also observe that many of the resulting subgraphs with known algorithms are highly overlapping. The best known mechanisms follow the basic scheme by Ullmann in [28], which is a recursive backtracking scheme. This scheme computes the solutions by incrementally enumerating and verifying partial solutions. Its worst case time complexity is $\Theta(|V|!\,|V|^2)$ [6]. This generic framework for the SQ problem is shown in Algorithm 1.

Algorithm 1. The algorithm begins with $S = \emptyset$ and recursively explores any matching for the query graph Q one node at a time. $cand(u)$, where $u \in V_Q$, is the set of vertices in V_G that have label $L_Q(u)$. Subroutine $IsJoinable$ determines if the edges between u and the already matched nodes in Q have matching edges between v and already matched vertices of G in S. If so, it updates the set S with the inclusion of v, and the newly matched edges. The recursion continues with $QSearch$; i.e., $QSearch$ is triggered within $QSearch$ to search for the next query vertex. When the recursive call returns, we restore S and continue with other candidates in $cand(u)$. $refineCandidate$ differs in different algorithms, ranging from simple degree checking to filtering by the neighborhood information of u. It is often the case that vertices that are near to each other are searched consecutively, which gives rise to many overlapping subgraphs in the solution.

Adapting Algorithm 1 for DSQ. A simple adaptation of this framework for DSQ is to consider all the candidate vertices for the first query node in the above framework and to try to retrieve embeddings in a random manner from these starting points. It is hoped that since the first candidates are dispersed, they may lead to diversified embeddings. However, our empirical studies show that this is not the case and the resulting coverage is not high. This is because although the first candidate vertices are distinct, the search paths may converge to common vertices in the remaining candidate matching. E.g. in Figure 1, let the first query node be $u3$, we may select $v7$ and $v9$ for $u3$. For the remaining searches, they may converge to $v3$ for $u1$, $v8$ for $u2$, and $v12$ for $u4$. Thus, the two embeddings returned will overlap at $v3$, $v8$, and $v12$.

2.3 Existing Maximum k-coverage Solutions

For DSQ, when all the embeddings (matchings) are given, the remaining problem is a special case of the maximum k-coverage problem with a subset size of $|Q|$. This well-studied problem is NP-hard, and we may adopt some known approximation algorithms.

GreedyDSQ. Given the set of all embeddings, GreedyDSQ keeps selecting the next embedding that gives the maximum gain in coverage, until k embeddings are obtained. GreedyDSQ has an approximation threshold of $(1-1/e) \approx 0.632$, which is the best possible guarantee for a polynomial time algorithm as shown in [12]. However, since the number of embeddings can be very large [1], the cost to generate and store all embeddings can be prohibitive.

Streaming Algorithms: SWAP0, SWAP1, SWAP2, SWAP$_A$. In the following, we describe four streaming algorithms which keep only a collection A_{curr} of k candidate embeddings at any time. The matching embeddings are scanned once, each newly scanned embedding may be swapped with an embedding in the current collection. The algorithms differ mainly in the swapping conditions. SWAP0 swaps the next embedding with an embedding in A_{curr} whenever it increases the coverage. However, this may result in a poor approximation. For a better guarantee we may swap embeddings based on the criterion in [25], resulting in algorithm SWAP1, or that in [3], resulting in algorithm SWAP2. Informally, SWAP1 swaps embedding s with a new s' when the benefit of adding s' is at least twice the loss of removing s in the coverage. SWAP2 swaps s with new s' when the coverage of the set of collected embeddings after swapping is $(1 + 1/k)$ times that of the current collection. Another algorithm SWAP$_A$ can be seen as a hybrid of SWAP1 and SWAP2, where a weighted combination of their swapping conditions is adopted [32]. Each of SWAP1, SWAP2, and SWAP$_A$ is a 0.25 approximate algorithm [25, 3, 32]. The streaming algorithms are still prohibitively costly since they require a feeding stream from the set of all embeddings. The preprocessing to generate all embeddings is required at query time and typically will take an excessive amount of time.

Use of SWAP2 in Diversified Clique Search. Diversified top-k maximal clique search is considered in [33]. This problem is similar to DSQ since diversity is also defined based on the coverage. [33] utilizes SWAP2 so that only a selection of k cliques is maintained at any time. With such a streaming algorithm, an initial set of k cliques need to be generated. The initialization step of [33] selects a set of k maximal cliques greedily for better pruning power. We also utilize a streaming algorithm. However, in our solution, the initialization step becomes a first phase, which is the more important phase since in most cases it returns the solution directly. A highly effective indexing, PNP, is proposed in [33] for the swapping phase. We adapt this index for DSQ in our implementation.

3. SOLUTION OVERVIEW

As discussed in Section 2.3, existing approximation algorithms for maximum k-coverage require the generation and scanning of all embeddings. Let us first assume that we can adopt an existing algorithm to generate all embeddings and that they can be stored and scanned. We shall remove this assumption later in this section. As pointed out in Section 2.1, if we aim for totally disjoint embeddings and succeed, we raise the approximation ratio to 1. Based on this idea, our first attempt is a simple multi-scanning approach starting with a zero overlapping constraint, and relaxing the overlap constraint level by level. We call this algorithm DSQ_{NS} (DSQ with No Swapping). It works as follows. We maintain a solution set T, and starting with $T = \emptyset$, we perform up to $|Q|$ scans of the embeddings. At the $i+1$-th scanning, a new embedding is selected if it contributes $|Q| - i$ new vertices to the coverage after this scan. DSQ_{NS} terminates when $|T| = k$. We denote the set of vertices in a set of embeddings S as $C(S)$. $|C(S)|$ is the coverage of S. Let us compare the solution of DSQ_{NS} with an optimal solution OPT. Clearly, $|C(OPT)| \leq k|Q|$. Suppose $|T| = k$ in the i-th scanning, then it is easy to see that $\frac{|C(T)|}{|C(OPT)|} \geq \frac{|Q|-i}{|Q|}$. If $i = 0$,

Algorithm 2: DSQL-P1 Framework: Phase 1 (no swapping)

```
1 begin
2     $i \leftarrow 0; T \leftarrow \emptyset$;
3     while $i < |Q|$ do
4         Generate a maximal set of embeddings $S$ s.t. for any $x \in S$,
          $|x \cap V(T)| = i$ and $\forall x, y \in S, x \cap y \subseteq V(T)$;
5         $T \leftarrow T \cup S$;
6         if $|T| \geq k$ then return $T$ and $i$;
7         $i \leftarrow i + 1$;
8     return $T$ and $i$;
```

T is optimal. Next consider the case where $|T| < k$ after the i-th scanning where $i = |Q| - 1$. In this case, each embedding o in OPT that is not chosen by the algorithm cannot contribute any new vertex to the solution T. That is, for each embedding o in OPT, it is either included in T, or all the vertices in o are already in $C(T)$. Thus, T is also optimal: $\frac{|C(T)|}{|C(OPT)|} = 1$.

While DSQ_{NS} has good approximate guarantee when it terminates early, there are two problems: (1) Typically, it is excessively costly to generate and scan all embeddings; (2) the worst case approximation guarantee is $1/|Q|$, which is below that of the swapping algorithms SWAP1 and SWAP2 if $|Q| > 4$.

We propose a novel level-based solution called DSQL (DSQ with Level-wise coverage) preserving the benefits of DSQ_{NS} while avoiding the above problems.

1. To avoid generating all embeddings, DSQL will selectively generate embeddings as the input to a mechanism that resembles DSQ_{NS}. Our study shows that the diversity objective can help to restrict the search scope significantly. The embedding generation process of DSQL is based on the recursive backtracking scheme of Algorithm 1.

2. To achieve a better guarantee than SWAP1, we propose a two phase algorithm. The first phase is non-swapping and resembles DSQ_{NS}. For the second phase we propose a multi-scan swapping algorithm called SWAP$_\alpha$. The embedding generation process remains level-wise and selective. The two phases are summarized in the following.

[**Non-swapping Phase**(DSQL-P1)]: The first phase of our solution is called DSQL-P1. The framework is shown in Algorithm 2. The solution set is collected in T, beginning with $T = \emptyset$. Initially we are at level 0. We collect a maximal set of disjoint embeddings in T and move on to level 1. Let T_i be the set of embeddings T immediately after level i. At level $i+1$, we try to add a maximal set of embeddings to T where each new embedding contains exactly $i+1$ vertices in $V(T_i)$, and all remaining vertices are distinct. When k embeddings are collected in T at any point at any level, the algorithm terminates, thereby simulating the effects of DSQ_{NS}.

[**Swapping Phase** (DSQL-P2)]: After Phase 1, to improve the approximate guarantee, we may continue with the second phase based on the swapping mechanism SWAP$_\alpha$. The swapping phase resumes the level-wise embedding generation, continuing at the level at which Phase 1 ends. Each generated embedding may swap with an embedding in T. Our swapping condition (Inequality (2)) allows us to set up an early termination criterion for this phase, which can significantly improve efficiency.

Some main properties of DSQL are summarized in the following.

LEMMA 1 (MAXIMALITY). *After the level i iteration in DSQL-P1, if $|T| < k$, then any embedding s that is not in T must overlap with the union of embeddings in T at $i + 1$ or more vertices.*

THEOREM 3. *Let OPT be an optimal solution. If DSQL-P1 stops at the i-th iteration then the result A has an approximation ratio of $\frac{|C(A)|}{|C(OPT)|} \geq \frac{|Q|-i}{|Q|} + \frac{i}{k|Q|}$, which is tight for any i and k. If it terminates at the $|Q| - 1$-th iteration and result A has size $|A| < k$, then A is optimal, i.e. $|C(A)|/|C(OPT)| = 1$.*

THEOREM 4. *Let OPT be an optimal solution. The approximation ratio of the solution SOL of DSQL is lower bounded by*

$$\frac{|C(SOL)|}{|C(OPT)|} \geq \max\left(\frac{1}{4}\left(1 + \frac{1}{k}\right), \frac{1}{4}\left(1 + \frac{1}{q}\right)\right)$$

We list below the meaning of some of the terms we use.

T	the current collection of embeddings
$V(T)$	set of vertices belonging to the embeddings in T
$candS(u)$	initial set of candidate vertices for u
$qList$	a list storing the rankings of query nodes
$Qovp$	set of query nodes that overlap with T
$ovpEmb$	a partial embedding (with -1 for non-overlap nodes)
$TcandS$	set of $candS(u) \cap T$ for $u \in V_Q$
$qfList$	list of ranked query nodes with father nodes
$Rcand$	candidate vertex sets as restricted by localized search
$pEmb$	a partial embedding as a set of (u, v) matching pairs

4. PHASE ONE OF DSQL: DSQL-P1

DSQL-P1 proceeds from level 0 up to level $|Q| - 1$, with possible early termination. A set T is used to collect embeddings. Algorithm 3 is the pseudocode for DSQL-P1.

Algorithm 3: DSQL-P1: the non-swapping phase

```
Input  : candidates sets $candS$, $k$
Output : embedding set $T$, $i$ (level number)
1 begin
      // Let $Q$, $G$, $T$, $candS$ be global variables
2     $qList \leftarrow Order(Q, candS)$;
3     $T \leftarrow \emptyset$;
      // $Q1Search$ updates $T$
4     $Q1Search(qList, \emptyset)$;
5     if $|T| == k$ then
6         return $T$, $k$;
7     foreach $i \in \{1, ..., |Q| - 1\}$ do
8         $QoverlapList \leftarrow \{$ $i$-subsets of $qList$ $\}$;
9         $TcandS \leftarrow \{ TcandS[u]$ from $T \mid u \in V_Q \}$ ;
          // $TcandS[u] = candS[u] \cap V(T)$
10        foreach $Qovp \in QoverlapList$ do
              // get $i$-overlap partial embeddings
11            while
              $ovpEmb = getNextOvpEmb(TcandS, Qovp)$ do
12                $Q1Search(qList, ovpEmb)$;
13                if $|T| == k$ then
14                    return $T$, $i$;
15    return $T$, $i$;
```

T is initially empty. New embeddings are added to T at each level. The value of i at Line 7 is the level number. When $|T| = k$, or when no more embeddings can be added, DSQL-P1 terminates. Let $V(T)$ be the set of vertices that belong to the embeddings in T. Let T_i be the value of T after the level i processing. At level i, a new embedding to be selected should overlap with $V(T_{i-1})$ at i vertices. To find an embedding, we first set i query nodes as **overlap nodes**, and the matching data vertices of these query nodes are selected from $V(T_{i-1})$. For the other query nodes, vertices in $V_G \backslash V(T_{i-1})$ are matched. Each vertex in $V_G \backslash V(T_{i-1})$ is matched at most once at level i. When such a new embedding is added into T, it brings $|Q| - i$ new vertices to T.

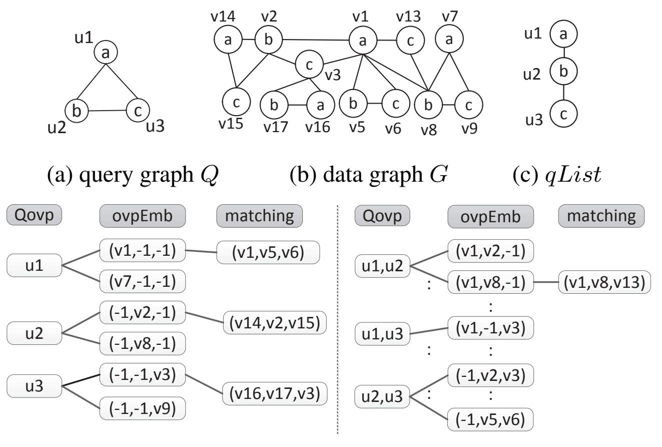
An index is pre-computed for looking up the set of vertices with a given label. We also use indices for **degree and neighborhood signature filtering** for candidates as in previous works such as [16]. Before running DSQL, we first generate a candidate set $candS(u)$ for each $u \in V_Q$ based on these filters. For better selectivity, DSQL ranks the query nodes and stores the ranking in a list $qList$ based on $|candS(u)|/degree(u)$, where $candS(u)$ is the set of data vertices with the same label as u, and $degree(u)$ is the degree of u in Q. DSQL-P1 next calls subroutine $Q1Search$ to get disjoint embeddings at Line 4 (we will describe $Q1Search$ in Section 4.1), resulting in T_0. If k embeddings are found, DSQL terminates. Otherwise, DSQL-P1 continues to search for embeddings with overlap (number of vertices that are already in current T) from size 1 to $|Q| - 1$ (Lines 7-14), in a level-wise manner. The algorithm terminates once k embeddings are found.

When the overlap size is i (Level i), each i-subset of V_Q is a possible set of overlap nodes, we denote such a set of query nodes as $Qovp$. Each $Qovp$ is sorted based on the ranking in $qList$, and is kept in $QoverlapList$ (Line 8). For each $Qovp$, DSQL-P1 finds matching data vertices for the overlap nodes in $Qovp$. For every query node $u \in V_Q$, we derive the candidate set $TcandS[u]$, where $TcandS[u] = candS[u] \cap V(T)$ (Line 9). For every $Qovp$, we find a match v for every node $u \in Qovp$ by picking a vertex in $TcandS[u]$, resulting in a valid **partial embedding**, $ovpEmb$, in which only the overlap nodes are matched (Line 11). These partial embeddings, $ovpEmb$, are passed to $Q1Search$ one by one to form any complete embedding. Let $qList = (u_1, ..., u_q)$, a partial embedding can be denoted by an ordered list of $(a_1, ..., a_q)$ where $a_i = v_i$ if v_i is the vertex matched to the overlap node u_i, and $a_i = -1$ (null) otherwise, i.e. -1 signifies an non-overlap node u_i that has not been matched.

4.1 Subgraph Search Function Q1Search

$Q1Search$ (Lines 4 and 12 in Algorithm 3) is similar to the recursive backtracking function of $QSearch$ in Algorithm 1 in Section 2.2, with three major differences:
(1) Some query nodes in $qList$ have been matched in the given partial embedding $ovpEmb$; there is no need to search for matchings for these nodes. The remaining nodes are searched in the order of $qList$; the candidate set for u is $candS(u) \setminus V(T)$.
(2) For each given $ovpEmb$, the only overlapping nodes should be the ones in $ovpEmb$. Once an embedding that matches all the null (-1) entries in $ovpEmb$ is found, there is no need to generate any more embeddings that match $ovpEmb$. This avoids the generation of many embeddings when compared to $QSearch$.
(3) Any data vertex matched in any full embedding is marked so that it will not be matched again for non-overlap nodes in other embeddings.

EXAMPLE 2. *Consider Figure 2. Let* $k = 6$. *Algorithm 3 ranks the query nodes in Q as qList* $= (u1, u2, u3)$. *First, it searches the candidates of* $u1$, *which are* $\{v1, v7, v14, v16\}$, *and get embeddings without overlap in* $T = \{(v1, v2, v3), (v7, v8, v9)\}$. *Since* $K = 6$, *and* $|T| = 2 < k$, *we continue to search for embeddings with overlapping vertices with* T. *We move on to Level 1 (see Figure 2(d)). The overlap size is set to 1; the ordered QoverlapList =* $(\{u1\}, \{u2\}, \{u3\})$. *From* T, *derive* $TcandS = (u1 : \{v1, v7\}, u2 : \{v2, v8\}, u3 : \{v3, v9\})$. *For each Qovp in QoverlapList, generate partial embeddings ovpEmb one by one and pass on to* $Q1Search$. *When* $Qovp = \{u1\}$, *we get ovpEmbs* $(v1, -1, -1)$, $(v7, -1, -1)$. *For ovpEmb =* $(v1, -1, -1)$, *we get embedding* $(v1, v5, v6)$. *After processing all entries in QoverlapList, T contains* $(v1, v2, v3)$, $(v7, v8, v9)$, $(v1, v5, v6)$, $(v14, v2, v15)$, *and* $(v16, v17, v3)$, $|T| = 5 < 6$. *We continue with Level 2 (see*



(a) query graph Q (b) data graph G (c) $qList$
(d) Level 1 (e) Level 2

Figure 2: (a) Q, **(b)** G, **(c)** $qList$, **(d)** $Qovp, ovpEmb$ **and embeddings generated at Level 1 of DSQL-P1, (e) some** $ovpEmb$**s at Level 2 if** $T = \{(v1, v2, v3)$, $(v7, v8, v9)$, $(v1, v5, v6)$,$(v14, v2, v15)$,$(v16, v17, v3)\}$.

Figure 2(e)) to search for embeddings with overlap size 2 ($i = 2$*). Now, QoverlapList =* $(\{u1, u2\}, \{u1, u3\}, \{u2, u3\})$, *and* $TcandS = (u1 : \{v1, v7, v14, v16\}, u2 : \{v2, v5, v8, v17\}, u3 : \{v3, v6, v9, v15\})$. $Q1Search$ *gets a new embedding* $(v1, v8, v13)$. *Now* $|T| = 6$, *so DSQL-P1 terminates. The results are* $(v1, v2, v3)$, $(v7, v8, v9)$, $(v1, v5, v6)$, $(v14, v2, v15)$, $(v16, v17, v3)$, *and* $(v1, v8, v13)$.

4.2 Indexing and Properties of DSQL-P1

DSQL-P1 requires the indices of $candS$, vertex degrees and neighborhood signatures for filtering candidates (see Line 3 of Algorithm 1). The **neighborhood signature** of a vertex v, denoted by $NS(v)$, is the set of labels of the neighbors of v. $NS(v) = \{\ell : (v, v') \in E$ and $L(v') = \ell\}$. The storage requirement is thus $O(|V| + |E|)$.

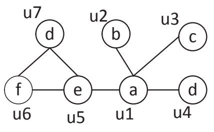
At each level i, Lemma 1 says that we generate a maximal set of new embeddings. An approximation guarantee for this phase is given in Theorem 3. The proofs are shown in the appendix.

5. OPTIMIZING DSQL-P1

The efficiency of DSQL-P1 can be greatly improved if we consider some properties of the diversity requirement. We introduce some optimization strategies in this section. The first two strategies prune partial embeddings by restricting the candidate sets for query nodes. The next two strategies avoid unnecessary subgraph search in the backtracking process. All four strategies require very little extra storage and are easy to implement.

5.1 Localized Subgraph Search

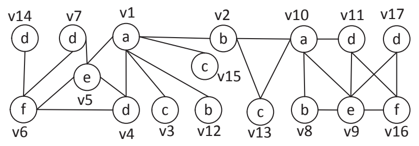
When we are given partial embeddings, some query nodes are already matched. We may greatly improve the performance by limiting the search scope to the neighborhood of the matched vertices. In Algorithm 3, we order the query nodes in $qList$ according to their selectivity. With this strategy, we reorder the vertices by giving higher ranks to matched nodes. We record essential information in a new data structure *qfList*. The elements of *qfList* are of the form (u, u_f), where u is a query node, and u_f is a designated **father node** for u. $u_f = -1$ for the first element in *qfList*. A father node u_f for node u is a query node that is processed before u and there is an edge linking u_f to u in the query graph Q. If (u, f) is the r-th entry in *qfList*, we say that the **rank** of u, $rank(u) = r$. We refer to u as *qfList*$[r].node$, and f as *qfList*$[r].father$.



(a) query graph Q

u	u1	u5	u6	u7	u3	u2	u4
u_f	-1	u1	u5	u5	u1	u1	u1

(b) *qfList*



(c) data graph G

Figure 3: Query graph Q, *qfList*, and data Graph G

EXAMPLE 3. *Figure 3 shows a query graph and its qfList. (The initial ordering in $qList$ is given by $u1, u5, u6, u7, u3, u2, u4$.) Suppose that currently $u1$ is matched to $v1$ in the data graph in Figure 3 (c). Q1iSearch in Algorithm 4 follows the order in qfList, thus, $u5, u4, u2, u3$ will be searched after $u1$. Since $u1$ is matched to $v1$, the search is localized in the neighborhood of $v1$. In qfList, the "father" of each of $u5, u4, u2, u3$ is $u1$. The candidates for $u5, u4, u2, u3$ will be limited to the neighbors of $v1$, which are $\{v5\}$, $\{v4\}$, $\{v2, v12\}$, $\{v3, v15\}$, respectively.*

This strategy is similar to that of candidate region exploration, which is a main idea proposed in [15]. However, compared to [15], we have a more powerful localization condition since we limit the search scope by T_0 based on the following lemma, which can be easily proved as a corollary of Lemma 1.

LEMMA 2. *For any level i, $i > 0$, let T_i be the value of T after Level i of DSQL-PL. Each embedding in $T_i - T_{i-1}$ must overlap with embeddings in T_0, and $|T_0| \leq k$.*

For this strategy, we introduce two subroutines as follows.

[**Subroutine reSort**($Q, qList, Qovp$)]
$reSort$ scans the nodes in $qList$ until the first matched node u ($u \in Qovp$) is found. $(u, -1)$ is entered as the first element in *qfList*. Next, for each neighbor u' of u in Q, if the father node of u' has not been set, it is set to be u, and (u', u) is added to *qfList*. After this, $(u, -1)$ is marked in *qfList*. We scan *qfList* until we come to the first unmarked element, and repeat a similar process as that for u. This is repeated until $|$*qfList*$| = |Q|$. Finally we shift the entries (u, u_f) to the end of *qfList* if $degree(u) = 1$ in Q.

[**Subroutine setCandidates**($u_j, u_f, pEmp, Qovp$)]
$setCandidates$ sets the candidate data vertices for a given query node u_j, given its father node u_f in *qfList*. Let the result be $Rcand$. First set $Rcand = candS[u_j]$. Next, do the following.

1. If u_f is matched to some vertex v_f in $pEmb$, set $Rcand$ to be the set of neighbors of v_f in G with the same label as u_j.
2. (1) If $u_j \in Qovp$: then further restrict $Rcand$ to vertices in T. Thus, $Rcand \leftarrow Rcand \cap V(T)$. (2) If $u_j \notin Qovp$: then, $Rcand \leftarrow Rcand \setminus V(T)$.

DSQL-P1 (Algorithm 3) is modified accordingly as follows. Function reSort is triggered initially after $qList$ is built at Line 2 and also in each level for each $Qovp$ after Line 10. We call the improved search function **Q1iSearch** (see **Algorithm 4**), passing the parameter of *qfList* instead of $qList$. We do not generate $ovpEmp$ at Line 11, but instead pass an empty set to *Q1iSearch* for the partial embedding. *Q1iSearch* will also search for the partial embedding matchings, $pEmb$. setCandidates is triggered in the beginning of *Q1iSearch* at Line 5, when a node u_j is to be matched and the father u_f is retrieved from *qfList*. In *Q1iSearch*, ***QSearchD*** is triggered when we encounter the first query node that is not in $Qovp$ (Line 16). $QSearchD$ is similar to *Q1iSearch* except that it returns true if an embedding is found and will not search for more embeddings, and $QSearchD$ will only trigger $QSearchD$ recursively.

5.2 Single Embedding Search Mode

Consider the example in Figure 2 again. If at level 1, $Qovp = \{u1\}$ and $ovpEmb = \{v1, -1, -1\}$, then suppose we assign a candidate $v5$ to match with $u2$. Next, we can only select a single embedding that contains $v1$ and $v5$, otherwise the overlap size will not be 1. Thus we enter the single embedding search mode with $pEmb = \{v1, v5, -1\}$. This search mode is handled by function $QSearchD$ in Algorithm 4. Recall that $QSearchD$ is triggered when we encounter the first query node that is not in $Qovp$. Since we only need to find a single embedding in the single embedding search mode, we further restrict the candidate set for query nodes. For this purpose we define $labelRm$ and $neighborRm$ for each query node when *qfList* is updated, the use of which will be explained shortly. $labelRm(u)$ is the number of higher ranked nodes in *qfList* that has label $L(u)$. $neighborRm(u)$ is the number of higher ranked nodes in *qfList* linked to u by an edge in Q. That is, $labelRm(u) = |\{u' : rank(u') > rank(u) \wedge L(u') = L(u)\}|$ $neighborRm(u) = |\{u' : rank(u') > rank(u) \wedge (u, u') \in E_Q\}|$.

query node	u1	u5	u6	u7	u3	u2	u4
labelRm	0	0	0	1	0	0	0
neighborRm	4	2	1	0	0	0	0

EXAMPLE 4. *The above table shows the values of $labelRm$ and $neighborRm$ for the example in Figure 3. For $u1$, no query node after $u1$ in qfList has label $L(u1) = a$, hence $labelRm(u1) = 0$. $neighborRm(u1) = 4$, since $u1$ is linked to $u5, u2, u3, u4$.*

For a query node u_j, if $neighborRm(u_j) = 0$, we randomly retain only $labelRm(u_j)+1$ valid vertices in $Rcand$, where a valid vertex is a vertex joinable to the matched vertices. In $QSearchD$, a candidate is matched if it is joinable to the vertices that are already matched. If $neighborRm(u_j) = 0$, the matching of u_j will not disqualify that for the higher ranked nodes in *qfList* due to this join criteria, so the matching for u_j can be randomly selected. The matching for u_j may affect other nodes with the same label, say u_k, since u_j may take up a good candidate for u_k. By choosing $labelRm(u_j)+1$ candidates we allow for one additional candidate that avoids such conflict. The enhanced $QSearchD$ has a stronger pruning power than $Q1Search$ since it restricts the candidate set and will not continue the search if no embedding is found after exhausting the candidates.

EXAMPLE 5. *Let us consider the case in Example 4. Since $neighborRm(u2) = 0$ and $labelRm(u2) = 0$, the number of candidates for $u2$ is limited to 1. In Figure 3, if $u1$ is matched to $v1$, according to qfList, the candidates of $u2$ are the neighbors of $v1$ with label b, namely, $v2, v12$. Hence, we randomly pick $v2$ to be the single candidate for $u2$, and $v12$ will not be matched. For*

$u7$, $neighborRm(u7) = 0$ and $labelRm(u7) = 1$, we pick 2 candidates from the neighbors of $v5$, namely, $v4$ and $v7$. We need to pick 2 candidates since if we pick $v4$ only, we cannot form the embedding with $v5$, for $v4$ should be matched to $u4$.

5.3 Skipping Query Nodes in Backtracking

For our next enhancement strategy, the idea is to skip the processing of certain query nodes under some condition to speed up the backtracking process in the $Q1iSearch$ or $QSearchD$ recursions. Let $pEmb$ be a set of matching pairs (u, v), where u is a query node, and v is a data vertex. When $pEmb$ contains a pair (u, v) for each vertex u in V_Q, it is a full embedding. In $Q1iSearch$ (Algorithm 4), $pEmb$ records the current partial embedding. $pEmp$ is initially empty on the first call of $Q1iSearch$, but grows as more matching pairs are found. Given two query nodes u_i and u_j, during the computation, when the current partial embedding is $pEmb$, we say that u_j **conflicts with** u_i if either of the followings holds:

1. u_j is linked to u_i by an edge in Q.
2. v_j has been matched to u_j in the current partial embedding, i.e. $(u_j, v_j) \in pEmb$. Also, v_j is a valid candidate for u_i, i.e. $L(v_j) = L_Q(u_i)$ and v_j passes the degree and neighborhood signature filters for u_i.

We use a **conflict table** to record the above conflict relationships. A conflict table is a boolean array of size $|Q|$. We construct a static conflict table for every $u \in V_Q$ as $CT(u, *)$, with an entry in $CT(u, *)$ for each query node. If u is connected with u' in Q, then the entry of u' in $CT(u, *)$ is set as true (1), otherwise it is set as false (0). When matching query node u_i, we construct a dynamic conflict table for u_i, $CT(u_i, \beta)$, where β is the current partial embedding, and first, we initialize all entries to false. If a candidate v passes the degree and neighborhood signature filter of u_i, but it is matched to u_j in β, then u_j conflicts with u_i, and we mark the entry of u_j of $CT(u_i, \beta)$ as true.

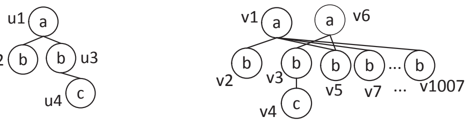
[**Skipping Non-conflict Nodes**]. We say that DSQL fails at u_i when it cannot find a vertex to match with u_i. If DSQL fails at u_i, it returns the conflict table $CT(u_i, \beta)$, where β is the current partial embedding. When backtracking to u_k, the algorithm checks tables $CT(u_i, \beta)$ and $CT(u_i, *)$. If the corresponding elements of u_k in both tables are false, then there is no conflict, and the algorithm skips u_k and backtracks to a higher level. Otherwise it continues with the next candidate for u_k.

[**Correctness**]. Suppose u_k is skipped by the strategy after failing at u_i and backtracking to u_k. This implies that u_k does not conflict with u_i, thus, u_k is not a neighbor of u_i in Q. Note that by the construction of *qfList*, the parent node of u_i will be encountered before its other ancestor nodes in the backtracking. Thus, the only way that the matching of u_k to a vertex v_k may affect the success or failure of matching u_i is when v_k is a valid candidate for u_i. This is because if we have not selected v_k for u_k, it can be matched to u_i, and may result in a success instead of a failure. However, since u_k does not conflict with u_i, v_k is not a valid candidate for u_i. Thus, we can safely skip u_k.

EXAMPLE 6. *Consider the example in Figure 4. The querying order for Q is u_1, u_2, u_3, u_4. $Q1iSearch$ matches v_1 to u_1 and v_2 to u_2, and gets partial embedding $\beta_1 = \{(u_1, v_1), (u_2, v_2)\}$. We continue to try to match u_3; candidates v_2, v_5, v_7 to v_{1007} all fail the degree filter test. So we fail at u_3, returning $CT(u_3, \beta_1)$. We backtrack to u_2. u_2 does not conflict with u_3 in both $CT(u_3, \beta_1)$ and $CT(u_3, *)$, so we directly backtrack to u_1 and use v_6 to match u_1. Note that without node skipping, the backtracking will process v_5, v_7,...,v_{1007} to map u_2, and fail to find any embedding. Thus, a large number of useless node searches are saved.*

Algorithm 4: Q1iSearch (to replace Q1Search in DSQL-P1)

```
Input  : $TcandS, qfList, start, Qovp, pEmb$
Output : $T$, $candS$, and $matched$ are updated
begin
    if $|pEmb| = |Q|$ then return;
    $j \leftarrow start$;
    $u_j \leftarrow qfList[j].node$, $u_f \leftarrow qfList[j].father$;
    $Rcand \leftarrow setCandidates(u_j, u_f, pEmp, Qovp)$;
    foreach $v \in Rcand$ following its ordering do
        if $u_j \notin Qovp$ and $matched[v] == true$ then
            continue;
        if $v$ fails to pass the degree or neighborhood filters then
            $candS[u_j] \leftarrow candS[u_j] \setminus \{v\}$; continue;
        if isJoinable($pEmb, u_j, v$) then
            $pEmb \leftarrow pEmb \cup \{(u_j, v)\}$;
            $matched[v] = true$;
            if $u_j \in Qovp$ then
                $Q1iSearch(TcandS, qfList, j+1, Qovp, pEmb)$;
            else if $QSearchD(TcandS, qfList, j+1, Qovp, pEmb)$ then
                $T \leftarrow T \cup pEmb$;
                if $|T| == k$ then return;
                continue;
            if $u_j \notin Qovp$ then $matched[v] \leftarrow false$;
            $pEmb \leftarrow pEmb \setminus \{(u_j, v)\}$;
```



(a) query graph Q (b) data graph G

u1	u2	u3	u4
1	0	0	1

(c) $CT(u_3, *)$

u1	u2	u3	u4
0	0	0	0

(d) $CT(u_3, \beta_1)$

u1	u2	u3	u4
0	1	0	0

(e) $CT(u_3, \beta_2)$

Figure 4: (a) Q, (b) G, and node conflict tables: (c) static conflict table (d) partial embedding is $\beta_1 = \{(u_1, v_1), (u_2, v_2)\}$ (e) partial embedding is $\beta_2 = \{(u_1, v_6), (u_2, v_3)\}$

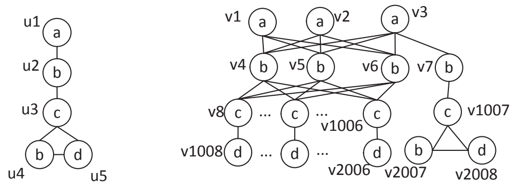
To continue, we match v_6 to u_1, and v_3 to u_3, The current partial embedding is $\beta_2 = \{(u_1, v_6), (u_2, v_3)\}$. We proceed to match u_3. However, the candidate v_3 is already mapped to u_2. Because of this, u_2 conflicts with u_3. After failing at u_3, we backtrack to u_2. Since u_2 conflicts with u_3, we cannot skip u_2, and match u_2 with the next candidate v_5. With further recursions, we get the embedding (v_6, v_5, v_3, v_4).

5.4 Skipping Data Vertices in Backtracking

When the average degree in the data graph is relatively high, there may be many vertices sharing the same or similar neighborhood. Failure with such vertices may incur many duplicated computations. [24] rewrites vertices with the same neighborhood as a super node, thus reducing the costs. Our algorithm simply keeps track of "bad" vertices to a similar effect.

Let the query nodes be processed in the order of $u_1, ..., u_q$, i.e., we assume the following ranking in *qfList*: $rank(u_1) < ... <$

$rank(u_q)$. Suppose we match a data vertex v_i for u_i, but we could find no match for u_{i+1}, then we mark v_i as a **"bad" vertex**. These markings of vertices are only kept for one layer: when we backtrack to the parent of u_i, all "bad" vertices marked at the matching for u_{i+1} are unmarked. When the subgraph search fails at query node u_i, we backtrack to the first query node u_j that conflicts with u_i. We examine the query node u_{j-1} before u_j. If u_{j-1} does not conflict with u_i, the data vertex for u_j is marked as a "bad" vertex. When we backtrack to the query node u_{j-1}, the processing of u_{j-1} will skip the "bad" vertices for the matching of u_j next time. We illustrate the effects of this strategy with the following example.



(a) query graph Q (b) data graph G

Figure 5: An example of a denser data graph

EXAMPLE 7. *Given G and Q in Figure 5. Consider the case when the partial embedding $pEmb = \{(u_1, v_1), (u_2, v_4)\}$ and we try to match u_3. The next candidate is v_8. Since there is no matching for u_4, the algorithm marks v_8 as "bad". Similarly, $v_9, ..., v_{1006}$ are marked as "bad" vertices. When the algorithm chooses v_5 for u_2, it will directly skip "bad" vertices, i.e. $v_8, ..., v_{1006}$. Then there is no matching for u_3, so v_5 is marked as "bad". The algorithm continues this process. Finally, when v_3 is mapped to u_1, v_5 and v_6 are all marked as "bad" vertices, the algorithm directly checks v_7 and finally finds one matching $(v_3, v_7, v_{1007}, v_{2007}, v_{2008})$.*

Note that v_{1006} will be marked as a "bad" vertex when there is no matching for u_4, as it is searched after matching v_6 to u_2. However, when we move up to u_1, v_{1006} will be unmarked.

LEMMA 3. *The strategy of skipping "bad" vertices is correct.*

A proof is given in the appendix. Note that this strategy and the previous strategy are also applicable for subgraph querying, SQ.

6. PHASE TWO OF DSQL: DSQL-P2

Phase two of DSQL continues the search from phase one, with the objective of enhancing the result by swapping embeddings and providing a better worst case approximate guarantee.

6.1 SWAPα: Multi-Scan with Swapping

To simplify our discussion, we assume again that we first have all the embeddings generated by an SQ mechanism. Then in Section 6.2 we shall remove this assumption. The streaming algorithms described in Section 2.3 are 0.25-approximate. In this subsection, we show that multiple embedding scannings can lead to better guarantees. The main results in this section are: (1) we propose a multi-scan algorithm SWAPα, and derive the parameter settings for progressive improvements on the approximation bound, which is asymptotically 0.5. (2) We introduce a new swapping condition in SWAPα which allows for early stopping. (3) We improve the approximation bound of the known algorithms of SWAP1, SWAP2, and SWAP$_A$ to $0.25 \times \max((1 + 1/k), (1 + 1/q))$, which also applies to SWAPα.

6.1.1 *Swapping Criterion for Each Scan*

Let $h = (V_h, E_h)$ be an embedding of G, $C(h) = V_h$, $|C(h)|$ stands for the coverage of h. Let $F = \{g_1, g_2, ..., g_k\}$ be a set of k embeddings of G, where $g_i = (V_i, E_i)$, $C(F) = \bigcup_i V_i$, $|C(F)|$ is the coverage of F. We assume that we are given a list of all embeddings. SWAPα consists of multiple passes, and in each pass, all embeddings are scanned once. Let the embedding list be $S = s_1, s_2, ..., |S| \geq k$.

In each pass, a collection F of k embeddings for the current best selections is maintained. We swap embeddings in the collection with a newly scanned embedding when the swapping criterion is satisfied. The first collection of k embeddings for the first pass is $F_0 = \{s_1, .., s_k\}$. The final collection generated by the t-th pass is the first collection of the $t+1$-th pass.

The coverage loss of an embedding f w.r.t. an embedding set F containing f is the coverage that is lost if f is deleted from F:

$$L(f, F) = |C(f) \setminus C(F \setminus f)| \tag{1}$$

The coverage benefit of an embedding h w.r.t. a set F of embeddings is the gain in coverage if h is added to F:

$$B(h, F) = |C(h) \setminus C(F)|$$

[**Swapping Criterion**] In general, we swap the next candidate h with any candidate $f^* \in F$ if for a certain parameter $\alpha \geq 0$,

$$B(h, F) \geq (1 + \alpha)L(f^*, F) \tag{2}$$

The loss function $L(f, F)$ above differs from the loss measurement in [25], which is given by $L^+(f, h, F) = |C(f) \setminus C(F \cup h \setminus f)|$. In our empirical studies, we show that the two loss functions result in similar performance in efficiency and quality for maximum k-coverage. However, $L(f, F)$ is independent of the new embedding h, which allows us to introduce an early stopping strategy in our algorithm for DSQ, as will be shown in the proof of Lemma 4 in Section 6.2.

6.1.2 *Progressive Gain with Multiple Scans*

The question is how to set the value of α in Equation (2) in each scan. Let α_t be the value of α used for the t-th scan of the embeddings. γ_t is a lower bound for the approximation ratio of the resulting collection of embeddings in the t-th scan. For a scanning of the embeddings, let F_0 be the initial embedding collection, and F_i be the embedding collection when f_i is removed from it, where f_i is the i-th embedding swapped out by the algorithm. Hence $f_i \in F_i$ and $f_i \notin F_{i+1}$. Let F_{final} be the final collection. We set γ_t as the value of $|C(F_{final})|/|C(OPT)|$ for the $t-1$-th scan or the value of $|C(F_0)|/|C(OPT)|$ for the t-th scan, where OPT is an optimal solution. Our main result on the setting of α and the coverage guarantee is the following:

THEOREM 5. *At the t-th scanning of SWAPα, if $\gamma_{t-1} < 0.5$, then by setting*

$$\alpha_t = 1 - 2\gamma_{t-1} \tag{3}$$

the approximation ratio of the embedding collection after the scanning is lower bounded by

$$\gamma_t = 0.25(1/(1 - \gamma_{t-1})) \tag{4}$$

A proof of Theorem 5 is given in the appendix. From Equations (3) and (4), if $\gamma_0 = 0$, $\alpha_1 = 1$, $\gamma_1 = 0.25$, $\alpha_2 = 0.5$, $\gamma_2 = 1/3$, $\alpha_3 = 1/3$, $\gamma_3 = 3/8$, $\alpha_4 = 1/4$, $\gamma_4 = 0.4$, $\alpha_5 = 0.2$, $\gamma_5 \approx 0.416$, $\alpha_6 = 1/6$, $\gamma_6 \approx 0.428$, $\alpha_7 \approx 0.114$, $\gamma_7 \approx 0.437$ ...

It can be shown that $\gamma_0, \gamma_1, ...$ converges to the fixed point of 0.5.

6.1.3 *A Better Bound for the Swapping Mechanism*

For the first scan, we can get a better bound if we do not simply pick the first k scanned embeddings for F_0. Instead, we begin with an empty F_0, and add the next scanned embedding h if the swapping criterion is met, assuming that an empty fictitious embedding f is swapped out. Since the loss $L(f, T) = 0$, the next embedding is added whenever there is non-zero additional coverage. This is repeated until k embeddings are collected in F_0. After that the algorithm continues as before. We refer to this as *the progressive initialization step*. We prove the following in the appendix.

THEOREM 6. *For DSQ, let SOL be the solution of any of the one-pass algorithms SWAP1, SWAP2, SWAP$_A$ and SWAPα with $\alpha = 1$, with the progressive initialization step,*

$$\frac{|C(SOL)|}{|C(OPT|)} \geq \max\left(\frac{1}{4}\left(1+\frac{1}{k}\right), \frac{1}{4}\left(1+\frac{1}{q}\right)\right)$$

E.g. if $k = 2$, $\gamma_1 = 0.375$, if $q = 5$, then $\gamma_1 = 0.3$.

This is a better bound compared to 0.25 derived in [25] and [32]

The result holds also for the maximum k-coverage problem when the given subsets are of the same size, q.

6.2 DSQL-P2

SWAPα assumes an input stream of embeddings. We now relax the seemingly necessary requirement to generate all embeddings. As in DSQL-P1, in DSQL-P2, we generate embeddings in a level-based approach and supply the embeddings to a SWAPα based algorithm, with an early stopping technique that can attain the same approximation guarantee without generating all embeddings.

After Phase 1 of DSQL, we obtain an embedding set T and also a level number i. T and i are input to the second phase, DSQL-P2. There are two possibilities:
(1) $i = |Q| - 1$ and $|T| < k$. From Theorem 3, the solution T is an optimal solution. Thus, we terminate without triggering Phase 2.
(2) Otherwise, $|T| = k$. This is because $|T| = k$ is the termination criteria at any level below $|Q| - 1$. There are two subcases:
(2a) The embeddings in T are disjoint. In this case, the algorithm has an optimal solution and is terminated.
(2b) Otherwise, if the approximation ratio of T is ≥ 0.5, return T as the solution. Else, trigger Phase 2, DSQL-P2.

We terminate at an approximation ratio[1] of 0.5 or above since the guarantee of SWAPα is bounded by 0.5. Note that at the beginning of Phase 2, $|T| = k$. DSQL-P2 is shown in Algorithm 5. The first step in DSQL-P2 is to save the input T as $T1$. In $Q2Search$, we generate embeddings as in the first phase except for a main difference: we always use $T1$ instead of T in the generation of $TcandS$. When an embedding h is found, we check if the swapping condition of Inequality (2) is satisfied for any embedding f in T: $B(h, T) \geq (1 + \alpha)L(f, T)$. If it is satisfied then we swap h with f. Next, we check if DSQL-P2 can be terminated. The termination condition is described below.

[**Early Termination**]: The swapping phase is terminated if both of the following two conditions hold:

(1) $V(T1) \subseteq V(T)$;

(2) For each embedding $f \in T$, $L(f, T) \geq (q - i)/(1 + \alpha)$.

If the above conditions do not hold, we continue with the generation of the next embedding. When all level i embeddings have been generated, and if $i < |Q|$, we continue with the next level of $j = i + 1$. This recursive process continues until either we can terminate early or $j = |Q| - 1$. The following lemma is proved in the appendix.

[1]The approximation ratio is taken to be $|C(T)|/kq$.

Algorithm 5: Swapping Phase: DSQL-P2(T,i)

```
Input  : $Q, G, candS, k, qList, T, i$ (level number)
Output : top $k$ embeddings ($T$)
1 begin
2     $T1 \leftarrow T$;
      // $Q, G, T, candS, k$ are global
3     foreach $j \in \{i, ..., |Q| - 1\}$ do
4         $QoverlapList \leftarrow$ {subset of $qList$ of size $j$};
5         $TcandS \leftarrow \{TcandS[u]$ from $T1 | u \in V_Q\}$;
6         foreach $Qovp \in QoverlapList$ do
7             while
              $ovpEmb = getNextOvpEmb(TcandS, Qovp)$ do
8                 if $\neg Q2Search(qList, 0, ovpEmb, T1)$ then
9                     return $T$;
```

LEMMA 4. *The early termination for DSQL-P2 is correct.*

Initially $T = T1$, therefore if for all $f \in T1$, $L(f, T1) \geq (q - i)/(1 + \alpha)$, the swapping process can be terminated. The overall approximation guarantee of DSQL is given by Theorem 4, a proof of which is included in the appendix.

7. EMPIRICAL STUDY

In this section, we present our experimental results. All our experiments are conducted on a machine with 3.4Ghz Intel Core i7-4770 CPU and 16 GB RAM, running Ubuntu 12.04 LTS Linux OS. All algorithms are implemented in C++. For existing algorithms we use the coding provided by the authors of [24] for BoostIso over $Turbo_{ISO}$, and for $Turbo_{ISO}$.

Datasets. We use 9 real datasets in our experiments: Human, Yeast, Youtube, Wordnet, DBLP, Epinion, USpatent, Dbpedia, and IMDB. The first six sets are used in [24], USpatent is used in [27], and DBpedia and IMDB are used in [18]. Dbpedia is an RDF graph crawled from Wikipedia[2]. We choose the person dataset and their links to build the graph. Epinion is a who-trust-who online social network[3]. The IMDB data set contains rich information of movies and TV series. We extract the relationship among movies, TV series, actors, actresses and directors to build the whole graph. Each of these datasets is one data graph. For Youtube, Epinion, DBLP, and Dbpedia, there are no given labels, as in [24], we have assigned a label for each vertex from a synthetic label set of sizes 100, 50, 50, and 100, respectively, with a uniform random distribution. The details are shown in Table1.

Query Set. Except for the special queries used in Section 7.2, we generate query graphs by randomly selecting connected subgraphs of G, using the query generator coding from the authors of [24]. There are 1000 query graphs in one query set with the same query size (the number of edges). The query size ranges from 1 to 10. Let the query size be z. The generator begins with an empty Q, and randomly picks a vertex u from G, puts it into Q, and continues to randomly choose an edge $e = (u, v)$ incident to a vertex u in Q from E, and adds v and e to Q, until there are z edges in Q. We vary k from 10 to 50. The default query size is 5 and the default k value is 40.

Measurements. In our experiments, we measure the runtime, which is by taking the average time per query after running 1000

[2]http://dbpedia.org/

[3]http://snap.stanford.edu/data/

Dataset(G)	$\|V\|$	$\|E\|$	$\|\Sigma\|$	Avg. degree
Yeast	3101	12519	31(184)	8.07
Human	4675	86282	90	36.92
Wordnet	76854	213308	5	5.55
Epinion	75879	405741	50^*	10.69
DBLP	317080	1.04M	50^*	6.62
Youtube	1.1M	2.9M	100^*	5.26
Dbpedia	809597	3.72M	100^*	9.19
IMDB	4.49M	7.49M	123	3.34
USpatent	3.77M	16.5M	388	8.75

Table 1: Statistics of datasets (* indicates synthetic labels)

random queries. We also measure the coverage, which is an average value taken over 1000 queries. Let A be a solution for DSQ. $|C(A)|$ is the coverage of A. If the optimal solution size, $|C(OPT)|$, is known, then the *approximation ratio* is given by $|C(A)|/|C(OPT)|$. Otherwise, the approximation ratio is set to $|C(A)|/kq$, where q is the query size in number of vertices, in which case, it is actually a lower bound on $|C(A)|/|C(OPT)|$. Again, we take the average value over the query set. Note that we set the time limit to 5 hours, if the algorithm cannot finish the 1000 queries, we would terminate the program.

7.1 Results with Existing Methods

Our first study computes the total number of embeddings for the real datasets when the query size is 5 and $k = 40$. We used the coding of BoostIso over $Turbo_{ISO}$ for generating the embeddings. The results are shown in Table 2. The average per query is taken over 1000 random queries for DBLP and 50 queries for the other datasets, since 50 queries already take many hours. We have no result for the remaining datasets as they take more than 5 hours for 50 queries. The numbers are very large except for DBLP, which is due to the smaller data size and average degree, and the more even distribution of labels in DBLP. This result shows that enumerating all embeddings leads to very large answer sets. In addition, the time to enumerate all embeddings is not scalable to large graphs.

	Yeast	Epinion	DBLP	Youtube	Others
average	123389.6	666387.4	412.3	36.3M	–
worst case	19.33M	1.02M	13559	1925.45M	–
time(sec)	121.75	11.28	1.32	775.46	–

Table 2: Total number of embeddings and query time by best known SQ method (per query) with $|E_Q| = 5$

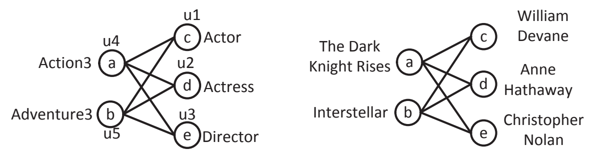
Next, we adopt a known SQ method and take the first k generated embeddings as the result. We apply the state-of-the-art algorithm of BoostIso on top of $Turbo_{ISO}$. The results are shown in Table 3. The coverages are small because the first k matchings are trapped in local areas, creating much overlapping. Thus, this approach cannot provide a well diversified solution.

	Yeast	Epinion	DBLP	Youtube	Others
$coverage$	21.05	21.76	39.07	25.82	–
approx ratio	0.105	0.091	0.168	0.108	–

Table 3: Results of best known SQ method, $|E_Q| = 5$, $k = 40$

7.2 Some Query Results with IMDB

Next we evaluate the performance of DSQL. We compare $DSQL$ with an interleaving search method adapted from a SQ solution. We call this method COM. We also apply our localized subgraph search and skipping strategies to this method. COM first sorts the query to form $qList$. Then for every candidate v_i of the first node in $qList$, it maintains a list of recursive iterators where each iterator traverses the candidates of every query node, and explores a search region rooted at v_i. COM processes the results in an interleaving manner. Once it finds an embedding in a search region rooted by v_i, it saves the states of the iterators for v_i, and then randomly jumps to another region rooted at another candidate v_j. When COM next jumps back to the search region rooted by v_i, it restores the states for continuing searching. COM terminates when k embeddings are found.



Query Graph A Subgraph Result

For IMDB, we build an edge if an actor/actress/director takes part in a movie/TV series. We label the movies and TV series by the rank and genre information, e.g., *Adventure3* stands for an excellent (rank $\geq$ 8.5) adventure movie. We examine the results of the query in the above figure, which resembles our motivating example. With the default setting of $k = 40$, COM gets a coverage of 97, while DSQL gets a coverage of 150. E.g. DSQL retrieves "Prison Break" for u_4, while COM does not. The figure above shows an interesting result returned by DSQL. Another interesting result is the following:
{Welliver_Titus, Scarwid_Diana, Beesley_Matt_Earl, Prison Break, Lost}

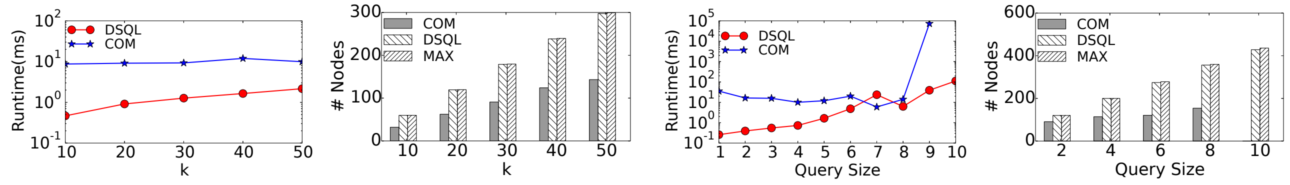
7.3 Performance of DSQL

In this subsection, we compare the runtime and coverage of DSQL and COM. For the comparison of each dataset, we vary the values of k from 10 to 50 and measure both the runtime and the coverage. We repeat this by varying the value of $|E_Q|$ from 1 to 10, following the settings of [24]. The results are shown in Figure 6. Here the number of nodes (# Nodes) refers to $|C(A)|$, where A is the resulting set of embeddings, for both DSQL and COM. To see how close the coverage is to the optimal result, we also plot a MAX value for comparison. With DSQL, we may discover an optimal solution if we get k disjoint embeddings or the number of embeddings in the solution is below k (see Theorem 3). In such cases, MAX is set to $|C(A)|$ if A is the solution set. Otherwise, MAX is set equal to $|V_Q| \times k$, which upper bounds the optimal coverage. Thus, we can compare the value of $|C(A)|$ for our solution A with MAX.

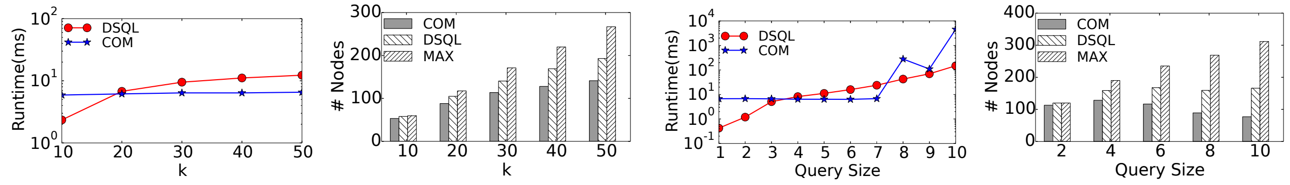
In general, the coverage increases with both k and $|E_Q|$, since more embeddings collected increase the number of covered vertices. Typically, a larger query size also leads to more covered vertices. COM covers much fewer vertices compared to DSQL. Even though it adopts an interleaving approach and randomly jumps between search regions, there is no mechanism to avoid overlapping as in DSQL. The coverage is generally very high for DSQL because it can avoid being trapped in a local area as it targets embeddings with limited overlapping.

For IMDB the label distribution is highly skewed, 90% of the labels are actor, actress or director, so the matchings can be highly numerous in some regions, and if a search enters such a region it can be very costly irrespective of the query size. Hence, we see some fluctuations in the runtime.

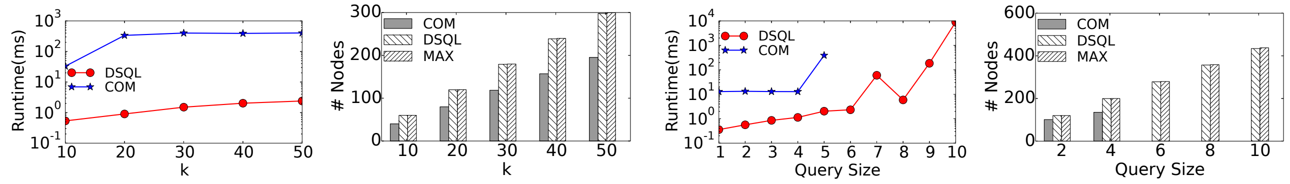
Since COM returns the first k embeddings found, it is quite fast when query size is small. When the data graph or query size is large, COM may not finish within our time limit of 5 hours. This happens with Wordnet when the query size is large than 9, and with IMDB when the query size is 6 or above. This is because COM may run into some deep recursion that involves many redundant computations without yielding any result. DSQL performs much



(a) query time and coverage for Wordnet



(b) query time and coverage for DBpedia



(c) query time and coverage for IMDB

Figure 6: Comparing DSQL with an interleaving search method, COM

better since the single embedding search stops searching when no match is found for a restricted set of candidates. DSQL also limits the exploration of search space within the single required embedding for each overlap pattern.

7.4 Effects of Varying the Label Set Size

We study the performance of DSQL with different label densities. Here we define label density as $\frac{|\Sigma|}{|V|}$. We consider datasets DBLP and Youtube with synthetic labels. For each graph, we vary the label density from $0.05 * 10^{-3}$ to $0.2 * 10^{-3}$.

In Figure 7, the bars labeled Youtube and DBLP show the coverages of DSQL algorithm, while MYoutube and MDBLP are their MAX values, respectively.

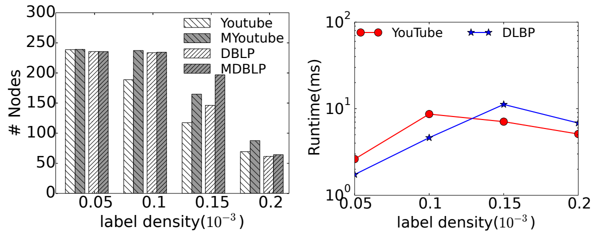


Figure 7: Effects of the label set size, k=40, $|Q|$=5

From the result, the coverage of DSQL is always close to MAX. As the label density increases, the approximation ratio would first decrease then increase, while the running time first increases then decreases.

The reason for these trends is the following. When label density is relatively small, there exist many matches in the data graph so that DSQL finds diversified results easily within a very short time. As the label density increases, there exists less matches. DSQL would terminate at higher levels, so the approximation ratio would decrease and more time is needed. As the label density grows even larger, there may not be enough k matches in the data graph, so DSQL often quickly terminates at the last level, and the coverage is close to MAX.

8. RELATED WORK

Subgraph isomorphism problem is an NP-complete problem[5]. A lot of effort has been devoted to solving it in a reasonable time for real datasets. Ullmann proposed the first practical algorithm for solving subgraph isomorphism problem [28]. It is a recursive backtracking algorithm which computes the solutions by incrementally enumerating and verifying partial solutions. In recent years, many works such as VF2[6], QuickSI[26], GraphQL[16], SPath[35] and TurboIso[15] have been proposed based on the backtracking framework. They improve the performance of the Ullmann algorithm by using vertex matching order strategies, adding powerful filters to prune invalid candidates early and choosing proper join orders. The best known complexity of an algorithm for this problem is that of VF2, which is $\Theta(|V|!\,|V|)$ [6], however, VF2 has been shown to be less efficient than variants of the Ullmann algorithm in later works. In [1], the number of subgraphs of G isomorphic to a given graph H is shown to be $O(|E_G|^{|V_H|})$ for finite simple graphs. BoostIso[24], TurboIso[15], and RBSub[11] solve the graph matching problem by graph compression. They exploit an equivalence relationship between vertices and the structure of the query graph to compress the data graph. Efficient RDF querying based on the properties of RDF data is studied in [19].

The indexing method that we use exploiting neighborhood information to prune candidate set of vertices has been used in previous works including GraphQL[16], SPath[35], and STwig[27]. The indexes are used as filters, so these algorithms are all under the backtracking framework in Section 2.2. An in-depth study is made in [21] comparing such methods and it is found that excessive indexing may lower the performance because of the overhead and limited filter effects. We adopt the best indexing strategy as noted in [21], which is that of the neighborhood signatures. For the study of querying a large set of relatively small graphs, indexes have been used as the filters of candidate graphs, some previous works are gIndex[30], FG-index[4], Tree+δ[36], and SwiftIndex[26]. A distributed algorithm STwig is proposed in [27] for solving subgraph isomorphism problem in billion-node graphs. Subgraph enumeration in MapReduce is considered in [20].

Approximate matching in large graphs is studied in SAPPER[34], where matching with bounded edit distance from the query graph is considered. [22] proposes the notion of strong simulation, with a tractable time complexity in the computation. Based on strong simulation, [11, 9] apply graph compression strategies to do efficient graph querying. [37, 10, 14] also study the problem of ranking the results. [37, 14] exploit the weights of nodes or edges, while [10] ranks the matched subgraphs based on their structure and diversity. [11] studies personalized social search in directed graphs where a query contains a particular person node.

The more general problem of diversity in search results has received much attention. A survey of top-k querying techniques in relational database systems is given in [17]. Three categories of diversity are identified in [7], namely, content based, novelty, and coverage. Algorithms of swapping heuristics and greedy selection have been used in diversifying recommendations for web tagging sites [31]. Efficient data access for diversity-aware search of relevant documents is studied in [2], and that for vector objects in [13]. A diversity measure based on distances among data objects is proposed in [8]. Redundancy-aware maximal clique searching is studied in [29]. Diversified top-k clique search proposes to find k maximal cliques in a data graph with the maximum coverage [33]. Another related work is [23], which reformulates a given query graph into a number of supergraphs to enrich the search results, and the search results are diversified.

9. CONCLUSION

We study the problem of diversified subgraph querying (DSQ) in a large graph, which is to find k subgraphs isomorphic to a given query graph with maximum coverage. We propose a novel level-based algorithm called DSQL with an approximation guarantee. DSQL proceeds from low to high levels. The level number refers to the number of common vertices of a newly selected subgraph with the collected subgraphs. Our experiments show that DSQL can generate highly diversified solutions with a quick response time.

10. ACKNOWLEDGEMENTS

We would like to thank the authors of [24] for kindly sharing their source code with us, Janet Yung for help in the proof-reading, and the reviewers for helpful comments. This research is supported in part by the Hong Kong RGC/GRF research grant 412313 Project ID 2150758.

11. REFERENCES

[1] N. Alon. On the number of subgraphs of presribed type of graphs with a given number of edges. *Israel Journal of Mathematics*, 38(1-1):116–130, 1981.

[2] A. Angel and N. Koudas. Efficient diversity-aware search. In *SIGMOD*, 2011.

[3] G. Ausiello, N. Boria, A. Giannakos, G. Lucarelli, and V. T. Paschos. Online maximum k-coverage. *Discrete Applied Mathematics*, 160(13-14):1901–1913, 2012.

[4] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, 2007.

[5] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[7] M. Drosou and E. Pitoura. Search result diversification. *SIGMOD Record*, 39(1):41–47, 2010.

[8] M. Drosou and E. Pitoura. Disc diversity: Result diversification based on dissimilarity and coverage. In *VLDB*, 2013.

[9] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168. ACM, 2012.

[10] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *VLDB*, 6(13):1510–1521, 2013.

[11] W. Fan, X. Wang, and Y. Wu. Querying big graphs within bounded resources. In *SIGMOD*, pages 301–312. ACM, 2014.

[12] U. Feige. A threshold of ln n for approximating set cover. *JACM*, 45(4):634–652, 1998.

[13] P. Fraternali, D. Martinenghi, and M. Tagliasacchi. Top-k bounded diversification. In *SIGMOD*, 2012.

[14] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han. Top-k interesting subgraph discovery in information networks. In *ICDE*, pages 820–831. IEEE, 2014.

[15] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337–348. ACM, 2013.

[16] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418. ACM, 2008.

[17] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[18] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: Fast graph search with label similarity. In *VLDB*, 2013.

[19] J. Kim, H. Shin, and W.S.Han. Taming subgraph isomorphism for RDF query processing. In *VLDB*, 2015.

[20] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. In *VLDB*, 2015.

[21] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *VLDB*, volume 6, pages 133–144. VLDB Endowment, 2012.

[22] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing topology in graph pattern matching. *VLDB*, 5(4):310–321, 2011.

[23] D. Mottin, F. Bonchi, and F. Gullo. Graph query reformulation with diversity. In *SIGKDD*, 2015.

[24] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *VLDB*, 8(5), 2015.

[25] B. Saha and L. Getoor. On maximum coverage in the streaming model and application to multi-topic blog-watch. In *SDM*, 2009.

[26] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *VLDB*, 1(1):364–375, 2008.

[27] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *VLDB*, 5(9):788–799, 2012.

[28] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[29] J. Wang, J. Cheng, and A. Fu. Redundancy-aware maximal cliques. In *KDD*. ACM, 2013.

[30] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, pages 335–346. ACM, 2004.

[31] C. Yu, L. Lakshmanan, and S. Amer-Yahia. It takes variety to make a world: Diverisification in recommender systems. In *EDBT*, 2009.

[32] D. Yuan, P. Mitra, H. Yu, and C. Giles. Updating graph indices with a one-pass algorithm. In *SIGMOD*, 2015.

[33] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. Diversified top-k clique search. In *ICDE*, 2015.

[34] S. Zhang, J. Yang, and W. Jin. Sapper: subgraph indexing and approximate matching in large graphs. *VLDB*, 3(1-2):1185–1194, 2010.

[35] P. Zhao and J. Han. On graph query optimization in large networks. *VLDB*, 3(1-2):340–351, 2010.

[36] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree+ delta<= graph. In *VLDB*, pages 938–949. VLDB Endowment, 2007.

[37] L. Zou, L. Chen, and Y. Lu. Top-k subgraph matching query in a large graph. In *Proceedings of the ACM Ph. D. workshop in CIKM*, pages 139–146. ACM, 2007.

APPENDIX

A. SOME PROOFS

Some theorems and lemmas are proved in this section.

A.1 Proof of Theorem 1

PROOF. We prove by showing that the decision problem of DSQ is NP-complete. In the decision problem of DSQ, we are given a data graph G, a query graph Q, and two values k and c. The question is whether there exists a set S of k embeddings of Q in G where $|C(S)| \geq c$. This problem is in NP since we can guess S and check its validity in polynomial time. It is NP-complete because we can transform the decision version of maximum disjoint isomorphic subgraphs, let us call it DMDIS (see Theorem 2), to this problem by first equating k in the DSQ problem to the required number of disjoint subgraphs in DMDIS, and then setting the value of c to $k|Q|$. $\square$

A.2 Properties of DSQL-P1

In DSQL-P1, let T_i be the currently collected set of embeddings, stored in T, after the $(i+1)$-th iteration. Note that T_0 contains disjoint embeddings. Let us restate Lemma 1:

LEMMA 1. *(restated) At the end of the i-th iteration of Algorithm DSQL-P1, $0 \leq i \leq q-1$, if $|T_i| < k$, then any embedding s that is not included in T_i must overlap with the union of embeddings in T_i at $i+1$ or more vertices.*

PROOF. We prove by induction. Consider the base case for the 1-st iteration. $T_0 = M$ is a maximal set of non-overlapping subgraphs. For any other embedding s, s must overlap with at least one of the embeddings in M, otherwise M would not be maximal, since s can be added to M. Thus, the base case holds.

At the beginning of $i+1$-th iteration, T_{i+1} is set to be equal to T_i. By the induction hypothesis, any remaining embedding s overlaps with T_i at $i+1$ or more vertices. Suppose s overlaps with T_i at vertices $w_1, ..., w_{i+1}, ...$ An $ovpEmb$ containing $w_1, ..., w_{i+1}$ will be generated in the $i+1$-th iteration (at Line 11 of Algorithm 3) and s is examined in Q1Search. There are 2 possible outcomes: (1) s does not overlap at any other vertex with the current T_{i+1}, it is included into T_{i+1}; (2) $|V(s) \cap V(T+i+1)| \geq i+2$. Since T_{i+1} grows monotonically, at the end of the iteration, s still has at least $i+2$ overlapping vertices with T_{i+1}. Thus, after the i-th iteration, all remaining subgraphs must overlap with T_{i+1} at $i+2$ or more vertices. $\square$

[**Proof of Theorem 3**]:

PROOF. Let $q = |Q|$. Clearly, $|C(OPT)| \leq kq$. Suppose DSQL-P1 stops at the i-th iteration and gets a solution A. If $|A| = k$, let n_j embeddings be added to the solution at iteration j for $0 \leq j \leq i$, we have $|A| = \sum_{j=0}^{i} n_j = k$. The first n_0 embeddings are disjoint and introduce qn_0 vertices. At the j-th iteration, the overlap size is j, every newly generated embeddings adds $q-j$ new vertices to $C(T_j)$, so we have $|C(A)| = qn_0 + \sum_{j=1}^{i} n_j(q-j) \geq qn_0 + \sum_{j=1}^{i} n_j(q-i) = qn_0 + (\sum_{j=1}^{i} n_j)(q-i)$. Since $n_0 \geq 1$, $\frac{|C(A)|}{|C(OPT)|} \geq \frac{q+(k-1)(q-i)}{qk} = \frac{q-i}{q} + \frac{i}{kq}$.

To prove that the above bound is tight for any i and k, construct a circle query Q with distinct labels and $|Q| = ik$. We set $q = ik$, so Q is a ring with $(u_1, u_2, ..., u_q)$. Let the data graph contain $k+1$ such circles, where one of the circle with vertices $(v_1, v_2, ..., v_q)$ overlaps at i unique vertices with other k circles. We denote this circle as C_0, and the other circles as $C_1, C_2, ..., C_k$. More specifically, $C_0 \cap C_i = \{v_{(i-1)q+1}, ..., v_{iq}\}$. Suppose DSQL first picks C_0 and then $C_1, ...C_{k-2}$, and returns T, then $|C(T)| = q + (q-i)(k-1) = (q-i)k + i$. Clearly the optimal solution OPT is $\{C_1, C_2, ..., C_k\}$, and so the approximation ratio is exactly $\frac{q-i}{q} + \frac{i}{kq}$.

If $|A| < k$, then after the $i = q-1$ iteration, DSQL-P1 still cannot get k embeddings. According to Lemma 1, any embedding s that is not included in A overlaps with the union of embeddings of A at q nodes. Let OPT be an optimal solution. In this case, each embedding o in OPT cannot contribute any new vertex to A. That is, for each embedding o in OPT, either $o \in A$, or all the vertices in o are already in A. Thus $\frac{|C(A)|}{|C(OPT)|} = 1$. $\square$

Next we show that DSQL-P1 does not miss any embedding since it effectively scans all embeddings in the given graph. Here, *effectively scanning an embedding h* means either including h in the solution, or dismissing h since it will not increase the coverage.

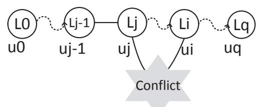
THEOREM 7. *After the $q-1$-th iteration, DSQL-P1 effectively scans all embeddings.*

Theorem 7 holds because Lemma 1 says that after the $q-1$-th iteration, any embedding that is not included in T_{q-1} must overlap with T_{q-1} at q vertices.

A.3 Proof of Lemma 3

Lemma 3 is about the correctness of the strategy of skipping "bad" vertices, which is introduced in Section 5.4.

PROOF. Consider the order of nodes in a query Q as shown in the figure below. The wavy line between u_0 and u_{j-1} means that some nodes may exist between u_0 and u_{j-1}.



Suppose we fail at u_i, this means that we cannot find a matching vertex for u_i. Let the nearest conflict node of u_i preceding u_i in $qList$ be u_j. Let the current partial embedding be
$pEmb_1 = \{(u_0, v_0), ..., (u_{j-1}, v_{j-1}), (u_j, v_j), ..., (u_{i-1}, v_{i-1})\}$.
We can see that for $j < k < i$, u_k is not a neighbor of u_i because u_k does not conflict with u_i. Based on the strategy in Section 5.3, we backtrack from u_i to u_j, skipping the non-conflict nodes. Suppose u_{j-1} is not a conflict node of u_i. We mark the candidate v_j for u_j as a "bad" vertex. Having tried v_{j-1} for u_{j-1}, we consider the next candidate v'_{j-1} and get $pEmb_2 = \{(u_0, v_0), ..., (u_{j-1}, v'_{j-1})\}$. The lemma says that we can skip any "bad" candidate v_j for u_j. We prove by contradiction that this holds. Assume on the contrary that this skipping is not correct, so that when we match v_j to u_j, we can eventually find a match v'_i for u_i, resulting in $pEmb_3$.
Let $pEmb_0 = \{(u_0, v_0), ..., (u_{j-2}, v_{j-2})\}$.
$pEmb_3 = pEmb_0 \cup \{(u_{j-1}, v'_{j-1}), (u_j, v_j), ..., (u_i, v'_i)\}$.
Let $pEmb_4 = \{(u_{j+1}, v_{j+1}), ... , (u_{i-1}, v_{i-1})\}$. Thus,
$pEmb_1 = pEmb_0 \cup (u_{j-1}, v_{j-1}) \cup (u_j, v_j) \cup pEmb_4$.

$pEmb_3$ shares with $pEmp_1$ the matchings in $pEmb_0$ and (u_j, v_j). Denote the set of neighbors of u_i which appear earlier than u_i in the query order as NS_{u_i}. So, we have $NS_{u_i} \subseteq \{u_0, u_1, ..., u_{j-2}, u_j\}$ = $\{u(pEmb_0) \cup u_j\}$, where $u(pEmb_0)$ is the set of query nodes that appear in $pEmb_0$. Since $(u_i, v'_i) \in pEmb_3$ and $pEmb_0 \cup (u_j, v_j) \subseteq pEmb_3$, then (u_i, v'_i) could also be used to extend $pEmb_1$. This is because all the neighbors of u_i are in $u(pEmb_0) \cup u_j$ and the other query nodes in $pEmb_1$ do not conflict with u_i. This contradicts the fact that there is no match for u_i when the current partial embedding is $pEmb_1$. $\square$

A.4 Proof of Theorem 5

We first introduce a lemma that will be useful.

LEMMA 4. *For a scan, assume that the coverage of the initial set of k embeddings, given by $|F_0|$, is lower bounded by $\gamma|C(OPT)|$, i.e., $|C(F_0)| \geq \gamma|C(OPT)|$.*

$$(2 + \tfrac{1}{\alpha} + \alpha)|C(F_{final})| \geq (1 + \tfrac{\gamma}{\alpha})|C(OPT)| \quad (5)$$

PROOF. Our proof is based on the concepts of set charge and element charge as used in [25] and [32]. The analysis is based on tracking the coverage $|C(OPT)|$ of an optimum solution OPT as embeddings in OPT are examined. For each embedding o in OPT, if o is not selected, a set charge is computed. If it is selected, then an element charge is computed for each vertex in o. We ensure that the total charge is an upper bound of $|C(OPT)|$.

[**Set Charge**] Let $\beta = 1 + \alpha$. For an embedding o_i in OPT, let H_i be the collection of k embeddings when o_i is examined. If o_i is not selected, then $\forall f \in H_i, B(o_i, H_i) < \beta \times L(f, H_i)$, hence

$$kB(o_i, H_i) < \beta \sum_t L(f_t, H_i)$$

It is easy to see that $\sum_t L(f_t, H_i) \leq |C(H_i)|$,

$$kB(o_i, H_i) < \beta|C(H_i)|$$

Set charge for each element in H_i is given by

$$\tfrac{B(o_i,H_i)}{|C(H_i)|} < \tfrac{\beta}{k}$$

Note also that $|C(F_{final})| \geq |C(H_i)|$.

Total set charge for all embeddings in OPT is less than

$$\sum_{i=1}^{k} |C(H_i)| \times \frac{\beta}{k} = \sum_{i=1}^{k} \frac{\beta|C(H_i)|}{k} \leq \beta|C(F_{final})|$$

[**Element Charge**] Let F_i be the embedding collection when f_i is removed from it, where f_i is the i-th embedding swapped out by the algorithm. Hence $f_i \in F_i$ and $f_i \notin F_{i+1}$. Let F_{final} be the final collection. The element charge is to keep track of the coverage of vertices in embeddings in OPT that have been selected, which may either appear in F_{final} or be swapped out. The total element charge is at most $|C(F_{final})| + \sum_{i\geq 0} |L(f_i, F_i)|$, where f_i is the i-th removed embedding.

$$\begin{aligned} |C(F_{i+1})| - |C(F_i)| &\geq B(h_i, F_i) - L(f_i, F_i) \\ &\geq (\beta - 1)L(f_i, F_i) \end{aligned}$$

$$\begin{aligned} \textstyle\sum_{i\geq 0} |L(f_i, F_i)| &\leq \tfrac{1}{\beta-1} \textstyle\sum_{i\geq 0}(|C(F_{i+1})| - |C(F_i)|) \\ &= \frac{1}{\beta - 1}(|C(F_{final}) - |C(F_0)|) \end{aligned}$$

[**Total Charge**] Summing up the set charges and element charges, we get $\beta|C(F_{final})| + |C(F_{final})| + \frac{1}{\beta-1}\,(|C(F_{final})| - |C(F_0)|)$

Clearly, this sum upper bounds $|C(OPT)|$. Since $\beta = 1 + \alpha$,

$$(2 + \tfrac{1}{\alpha} + \alpha)|C(F_{final}) - \tfrac{1}{\alpha}|C(F_0)| \geq |C(OPT)|$$

If the coverage of the initial set of k embeddings is lower bounded by $\gamma|C(OPT)|$, i.e., $|C(F_0)| \geq \gamma|C(OPT)|$, we have

$$(2 + \frac{1}{\alpha} + \alpha)|C(F_{final})| \geq (1 + \frac{\gamma}{\alpha})|C(OPT)|$$

□

[**Proof of Theorem 5**]: From Inequality (5) in Lemma 4, we have

$$\frac{|C(F_{final})|}{|C(OPT)|} \geq \frac{1 + \frac{\gamma}{\alpha}}{2 + \frac{1}{\alpha} + \alpha} = \frac{\alpha + \gamma}{(\alpha + 1)^2} \quad (6)$$

Differentiate w.r.t. α and set the result to zero, we get an optimal value for α for the swapping condition.

$$\alpha = 1 - 2\gamma \quad (7)$$

From Equations (6) and (7),

$$\begin{aligned} \alpha_{t+1} &= 1 - 2\gamma_t \\ \gamma_{t+1} = \frac{\alpha_{t+1} + \gamma_t}{(\alpha_{t+1} + 1)^2} &= \frac{1}{4(1 - \gamma_t)} \end{aligned}$$

□

Similar proof arguments will show that a similar theorem applies with $L^+(f, h, F)$ replacing $L(f, F)$. Note that in [25], α_1 is set to 1, which follows Equation (3) in Theorem 5, assuming $\gamma_0 = 0$.

At the fixed point of Equation (4), $4\gamma(1 - \gamma) = 1$. Solving for the equation of $4\gamma^2 - 4\gamma + 1 = 0$, we obtain the fixed point value of $\gamma_\infty = 0.5$. If $f(x) = \frac{1}{4(1-x)}$, $f'(x) = \frac{1}{4(1-x)^2}$, and $f'(0.5) = 1$. Thus the sequence of $\gamma_0, \gamma_1, \ldots$ converges to the fixed point of 0.5.

A.5 Asymptotic Tightness of SWAPα

Consider an online model of DSQ where embeddings are released one at a time. An online algorithm under such a model is allowed to keep at most k candidate embeddings at any time. We say that an online algorithm is greedy if the coverage of the collected embeddings can only increase.

LEMMA 5. *Any deterministic greedy online algorithm for DSQ cannot have an approximation ratio guarantee above 0.5.*

PROOF. Let the embedding size be Δ. Suppose the adversary first generates the following sequence of embeddings, $G_1, \ldots, G_{k''}$, where each G_i contains a common subset of R and also a subset X_i that is distinct for each G_i. $|R| = \Delta - 1$, $|X_i| = 1$.

Suppose that the algorithm keeps $k' \leq k$ such embeddings, let the embeddings be $R \cup A_1$, $R \cup A_2$, ..., $R \cup A_{k'}$, and dismisses embeddings $R \cup B_1, \ldots, R \cup B_j$, $j = k'' - k'$.

There must be a point when $j \geq k - \lceil k'/\Delta \rceil$. At this point, the adversary submits the following embeddings: $A_1 \cup A_2 \ldots \cup A_\Delta$, $A_{\Delta+1} \cup \ldots \cup A_{2\Delta}, \ldots$.

The optimal solution covers $A_1, \ldots, A_{k'}, R, B_1, \ldots, B_{k-\lceil k'/\Delta \rceil}$, the coverage is $\Delta - 1 + k' + k - k'/\Delta$ $= \Delta - 1 + k'(1 - 1/\Delta) + k$.

The solution from the algorithm covers $R, A_1, A_2, .., A_{k'}$, the coverage is $\Delta - 1 + k'$. Approximation ratio $\gamma = \frac{\Delta-1+k'}{\Delta-1+k'(1-1/\Delta)+k}$. This function increases with the value of k', and $0 \leq k' \leq k$. Thus, we have the greatest ratio when $k' = k$.

If $k' = k$, $\gamma = \frac{\Delta-1+k}{\Delta-1+k(2-1/\Delta)}$, which approaches 1/2 if k is large. □

A similar proposition is proved in [3], however their proof requires that Δ is a multiple of $\sqrt{k} + 1$, i.e., $\Delta/(\sqrt{k} + 1) \in \mathbb{N}$, which may not hold in our case since our query size, Δ, can be smaller than $\sqrt{k} + 1$. Our proof avoids this assumption.

For the case of multiple scans, if the only information that is passed from one scan to the next is the solution set, then this can be seen as passing the embeddings in the solution sets as the prefix in the embedding stream to the next scan. For the example used in the above proof, the solution contains only embeddings of the first type, namely $R \cup A_i$. Thus, arguments in the above can be used to show that for any greedy multi-scan online algorithm, the best approximation ratio guarantee approaches 0.5 as k is large.

A.6 Proof of Theorem 6

PROOF. Consider the algorithm of SWAP$_\alpha$. If F_0 is returned by the progressive initialization step, clearly, $|C(F_0)| \geq q + k - 1$. Since $|C(OPT)| \leq kq, q \geq |C(OPT)|/k$ and $k \geq |C(OPT)|/q$. We have $|C(F_0)| \geq \max(|C(OPT)|/k, |C(OPT)|/q)$. Thus, $\gamma_0 \geq \max(1/k, 1/q)$, and Inequality (6) follows from Theorem 5.

Since the initialization is altered, we need to prove that we get the same result as the original algorithms with a certain input stream. Let $F_0 = \{f_1, f_2, ..., f_k\}$ after the progressive initialization step. Let $g_1, ..., g_t$ be the embeddings that are dismissed in the initialization. Let F^i be the collection when g_i is examined. Clearly, $B(g_i, F^i) = 0$, since otherwise, the swapping condition is satisfied as the loss of swapping out a fictitious empty embedding is 0. Hence, $C(g_i) \setminus C(F^i) = \emptyset$. Since $C(F^i) \subseteq C(F_0)$, $C(g_i) \setminus C(F^i) = \emptyset$, thus, $B(g_i, F_0) = 0$. Therefore, F_0 is also obtained after $f_1, ..., f_k$, and $g_1, ..., g_t$ are scanned if the algorithm collects the first k embeddings, and the input stream begins with $f_1, ..., f_k, g_1, ..., g_t$.

Similar results can be readily derived for SWAP1, SWAP2, and SWAP$_A$. In particular, that for $SWAP_A$ can be derived by setting $\beta = \beta_1 + \beta_2$, $\beta_1 = (1 - \alpha_i)$ and $\beta_2 = (1 + \alpha_i)$ in Inequality (23) in [32]. □

A.7 Proof of Lemma 4

PROOF. We need to show that early termination does not affect the result. In the following, *effectively scanning* an embedding h means either including h in T, or dismissing h since it does not qualify for swapping. Let DSQL-P2 begin at level i. Similar to Lemma 1, we can show that at the end of the level j iteration of Algorithm DSQL-P2, $j \geq i$, any embedding that has not been effectively scanned must overlap with $V(T1)$ at $j + 1$ or more vertices. Thus, at level i, the overlap size with $T1$ for any newly scanned embedding h is at least i. If $V(T1) \subseteq V(T)$, then for any new embedding h, the benefit of h is at most $q - i$, i.e., $B(h, T) \leq q - i$. If in the current T, for each embedding $f \in T$, $L(f, T) > (q - i)/(1 + \alpha)$, hence, $B(h, T) < (1 + \alpha)L(f, T)$. Thus, there will not be any new embedding with a benefit that satisfies the swapping criteria of Inequality (2). Hence DSQL-P2 can be terminated without affecting the result. □

A.8 Proof of Theorem 4

PROOF. With DSQL-P2 we consider the swapping for each embedding with possible gain in the coverage. The algorithm is the same as SWAP$_\alpha$ except that we are given an initial set of k embeddings from DSQL-P1, and early termination may take place. From Lemma 4, we know that the early termination does not change the result. It remains to show that the initial collection F of k embeddings from DSQL-P1 is a possible set of first k embeddings for F_0 as constructed by the process described in Section 6.1.3. This is true if for each embedding s added to T in DSQL-P1, there is non-zero benefit $B(s, T)$. This is guaranteed by DSQL-P1, since when an embedding s is added to T at level i in DSQL-P1, $|Q| - i$ vertices in s are not in $V(T)$, and $i \leq |Q| - 1$. Thus, DSQL has the guarantee as stated in Theorem 6. □

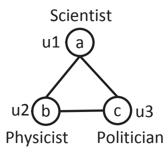
B. MORE EXPERIMENTAL RESULTS

In this section, we include some additional experimental results.

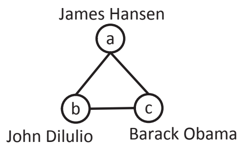
B.1 Some Query Results with DBpedia

For DBpedia in this experiment, we extract the occupation information from the dataset and use it as the label for every person vertex. We obtain 195 major occupations from the data and name the remaining occupations as *Other*. So there are 196 distinct labels in total.

Assume that we are interesting to find politicians who are connected with scientists and physicists, thus we submit the following query graph for DBpedia.



query graph



a result subgraph

For this query, with $k = 40$, DSQL obtains the above result subgraph that is about the current US President. Some other interesting results are:

{{Anatoli_Blagonravov}{Thomas_O._Paine}{Richard_Nixon}}
{{Michael_Faraday}{Joseph_Priestley}{Richard_Sharp }}

Note that Richard Nixon was the US President presiding over the Apollo 11 moon landing. Richard Sharp was a British Member of Parliament that was known for founding the London Institute.

B.2 Comparison of Swapping Strategies

For the comparison of the swapping strategies, we first generate all embeddings for DBLP. We tried two generators: using the coding of BoostIso [24] on TurboIso [15], and that of TurboIso provided by the author of [24]. The results are shown in Table 4. We apply the greedy algorithm GreedyDSQ (Greedy in the table) and also different swapping algorithms (single scan) on the set of all embeddings. For swapping and GreedyDSQ, t is the time for generating the embeddings. GreedyDSQ takes more time since it requires k scans, and the coverage is a little better compared to the other swapping algorithms. Our swapping condition with SWAPα has similar diversity result compared to other swapping conditions. BoostIso is more efficient than TurboIso (with runtime t), with slightly smaller coverages. Also, if we compare the coverages with the results in Table 3, we can see that applying maximum k-coverage techniques can greatly improve the diversity. Finally, DSQL has the best results in both time and coverage.

	SWAP1	SWAP2	SWAP$_A$	SWAPα	Greedy	DSQL
time (ms)	22.68+t	26.57+t	3.31+t	9.03+t	251.61+t	10.06
coverage	114.76	115.56	112.57	114.64	118.42	127.4

(a) results of running BoostIso on TurboIso ($t = 116.00$)

	SWAP1	SWAP2	SWAP$_A$	SWAPα	Greedy	DSQL
time (ms)	165.11+t	170.06+t	8.36+t	28.02+t	451.02+t	10.06
coverage	121.66	122.55	119.85	119.95	127.5	127.4

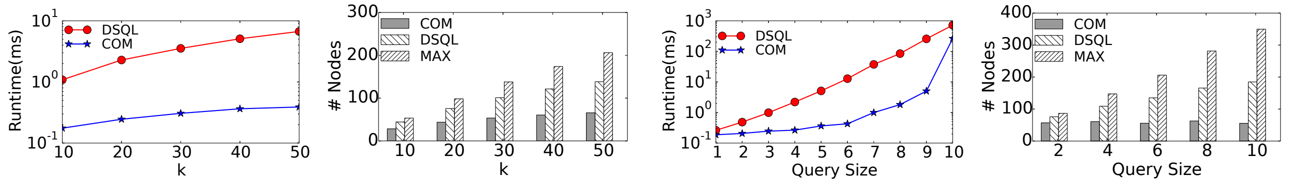
(b) results of running TurboIso ($t = 217.95$)

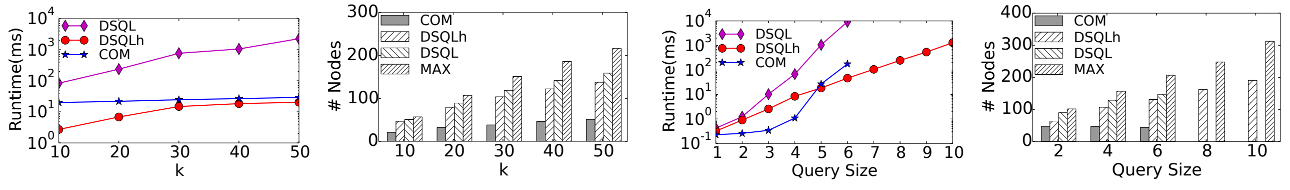
Table 4: Comparing the time and coverage of GreedyDSQ, swapping algorithms, and DSQL for DBLP, $|E_Q| = 5$, $k = 40$

We have compared the coverage results from multiple scans. The results show that the coverage improvement is not big with additional scans. Note that the approximation ratios are above 0.5, the asymptotic theoretical bound. Also note that in our experiments with DSQL (see Section 7.3), there is little scanning in DSQL-P2 due to early termination.

B.3 More Results on Comparison with COM

We show the results for the datasets of Yeast, Human, and USpatent in Figure 8. We compare our proposed method with COM, the interleaving method described in Section 7.3 in terms of query time and coverage in the number of nodes.


(a) query time and coverage for Yeast


(b) query time and coverage for Human

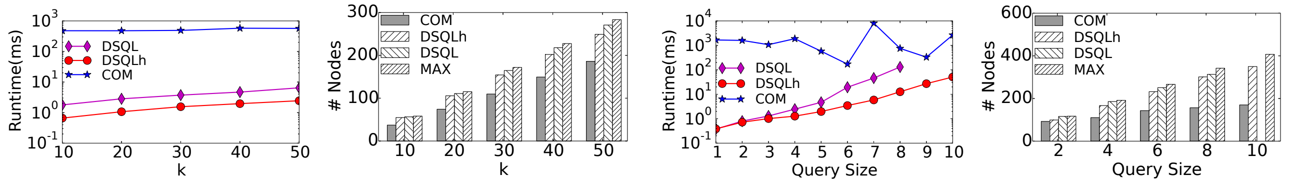

(c) query time and coverage for USpatent

Figure 8: Comparing the performances of DSQL, DSQLh, and COM

The trends are similar to that for the other datasets reported in Section 7.3. COM runs faster for smaller query and dataset sizes, but becomes inefficient as query size increases or dataset is large.

For Human and USpatent, both DSQL and COM cannot finish the query batches of 1000 queries of large query sizes within 5 hours, we thus introduce a variation of DSQL, called DSQLh. It differs from DSQL in the strategy of skipping bad vertices. We try to match u with its candidates. (1) If we cannot find any match for u, then as in DSQL in Section 5.4, we fail at u and backtrack to u_c, the closest conflict node of u. If u_{c-1} does not conflict with u, we mark the corresponding matched vertex v_c as "bad". (2) If we can find matches for u, but the matches are "bad" vertices, then we regard the matching of u as a failure, and begin the backtracking process of u as in (1). This deviates from DSQL where we would check the node u_p preceding u in *qfList*, and if u_p does not conflict with u, we mark matched v_p as "bad", and try to match u_p with the next vertex. With this variation, more skipping is allowed and there is more impact by "bad" vertices. A smaller coverage may be returned, but it can be much more efficient for denser data graphs. Our results show that DSQLh is efficient with both Human and USpatent. In summary, DSQL returns a solution within 10ms on average in most datasets, the coverage of DSQL is close to MAX and is much higher than that of COM.

B.4 Different Strategies of DSQL

Next we study the effects of the proposed optimization strategies. The following variations of DSQL are evaluated:

◇ DSQL0: Most primitive method, only using the localized subgraph searching strategy (see Section 5.1).

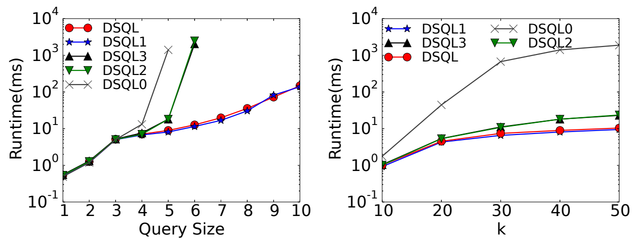
◇ DSQL1: Combining DSQL0 and the single embedding search strategy using $labelRm$ and $neighborRm$ (see Section 5.2).

◇ DSQL2: Combining DSQL0 and the conflict table strategy in Section 5.3.

◇ DSQL3: Combining DSQL2 and the "bad" vertex skipping strategy in Section 5.4.

◇ DSQLh: The variation of DSQL equipped with the relaxed skipping strategy introduced in Section B.3.

Still we set the time limit for 1000 queries to 5 hours. The default setting is a query size of 5, and $k = 40$. The results in Figure 9 show that every strategy can help reduce the runtime, since DSQL0 has a much longer runtime in comparison. DSQL1 is almost as good as DSQL in Figure 9(a). The single embedding search is effective because it controls the overlap size and avoids getting trapped in a local search area.


(a) query time for Youtube

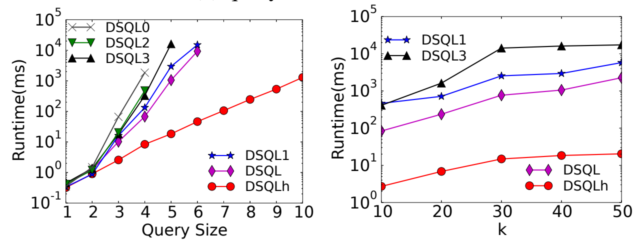

(b) query time for Human

Figure 9: Effects of the optimization strategies

From Figure 8(a), DSQL2 and DSQL3 are not as effective as DSQL1 for Youtube. The skipping strategies are useful for denser graphs. This is demonstrated by the results with Human in Figure 9(b). DSQLh, based on DSQL2 and DSQL3, greatly reduces the runtime while the coverage is still close to MAX (see Figure 8(b)). In the last graph, DSQL0 and DSQL2 are not shown since the runtime is too long.