Efficient Answering of Why-Not Questions in Similar Graph Matching

Md. Saiful Islam, *Member, IEEE,* Chengfei Liu, *Member, IEEE,* and Jianxin Li

Abstract—Answering *why-not* questions in databases is promised to have wide application prospect in many areas and thereby, has attracted recent attention in the database research community. This paper addresses the problem of answering these so-called *why-not* questions in similar graph matching for graph databases. Given a set of answer graphs of an initial query graph q and a set of missing (*why-not*) graphs, we aim to modify q into a new query graph q^* such that the missing graphs are included in the new answer set of q^*. We present an approximate solution to address the above as the optimal solution is NP-hard to compute. In our approach, we first compute the bounded search space and the distance to be minimized for q^*. Then, we present a two-phase algorithm to find the new query q^*. In the first phase, we generate a set of candidate edges to be added/deleted into/from the initial query q within the bounded search space and in the second phase, we select a subset of candidate edges generated in the first phase to minimize the distance for q^*. We also demonstrate the effectiveness and efficiency of our approach by conducting extensive experiments on two real datasets.

Index Terms—Graph Distance, Similar Graph Matching, Why-not Questions and Graph Query Refinement

1 Introduction

TODAY'S database systems are highly efficient in terms of query execution and resource usage. However, these systems are not usable for the end users to the same degree as they are proficient in underlying data management [1]. These days users are not satisfied with receiving the query output only, they also want to know why the system returns the current answer set as the output. In particular, they want to know *why* the system did *not* return an expected data object in the current answer set. Answering this kind of *why-not* questions in database systems has arisen as a natural demand for gaining the trust of the end users and is also promised to have real application potential in many areas including explaining missing answers in relational databases [2], [3], top-k queries [4] and reverse skylines [5].

Graph data management and matching similar graphs are predominant in a wide variety of applications including bioinformatics, computer vision, VLSI design, bug localization, road networks, social and communication networking. There are many indexing and query processing techniques that have been proposed for managing and querying graph data [6], [7], [8], [9]. In similar graph searching, a user is returned with either the top-k database graphs that are most similar to the query graph, $TOP_k(q, D)$ [8] or database graphs whose distances with the query graph are below a threshold, δ [9] with respect to a given matching policy. In such query settings, a user may not receive certain database graphs that are very similar to the query graph if the query graph is inappropriate/imperfect for the expected answer set.

• *Md. Saiful Islam and Chengfei Liu are with Swinburne University of Technology, Australia. Jianxin Li is with RMIT University, Australia. E-mail: {mdsaifulislam, cliu}@swin.edu.au, jianxin.li@rmit.edu.au*

To exemplify the above, consider a drug designer who is looking for chemical compounds that could be the target of her hypothetical drug before realizing it. In response to her query, the system can return the chemical compound structures from the database that are most similar to the query graph by applying the method explained in [9], [8] (structurally similar compounds have similar biological activity [10] and structural similarity search among compounds is a standard tool used in in-silico drug discovery [11]). However, it may not be always possible for her to formulate a query for a chemical compound structure that can perfectly match her need. As a result, the answer set may miss some expected targets. In this case, she may seek assistance from the system for modifying her hypothetical query graph so that it can include her missing targets in the new answer set: *"Is there any other query graph better than the current one for matching the expected answer set?"*. Consider another example from software clone detection area [12]. A developer is looking for the similar implementations of a program by issuing a query for her own implementation to the the database of *program dependency graphs* (PDGs). Again, she may miss some expected similar implementations which are known to be clones of her own program. She may wonder: *"Is my program is too dissimilar to the missing programs?"*. She may then ask the system to identify what changes could possibly make her program (i.e., modification of the program) to be the clone of those stored in the database. However, the new query must minimize the distance with the expected answer set to adhere to the notion of similarity matching (too distant objects are not similar at all). Here, we study this kind of problem of answering *why-not* questions (missing graphs) in similar graph searching/matching for graph databases formulated as given as below.

Given a set of graphs, $X = \{g_1, g_2, ..., g_k, g_{k+1}, ..., g_n\}$, where $X_1 = \{g_1, g_2, ..., g_k\}$ is the set of graphs returned by the underlying system in response to the user query graph q and $X_2 = \{g_{k+1},\ g_{k+2}, ...,\ g_n\}$ is the set of expected or missing graphs in the current answer set of q. We assume that the graphs in the set X_2 are not too distant from q. Now, our goal is to modify the initial query graph q into a new query graph q^* such that the missing graph set X_2 is included in the new answer set of q^* along with X_1. To do so, we need to minimize the distance between the new query q^* and the graph set X to adhere to the notion of similarity matching.

The optimal solution to the above problem corresponds to finding the maximum common subgraph (*mcs*) between the query graph q and the graph set X (i.e., subgraph common to q and each $g_i \in X$), computing the distances between the query graph q and each graph $g_i \in X$ (i.e., number of mismatched edges between q and g_i), and selecting an optimal subset of candidate edges from $V(U) \times V(U)$, where $V(U) = V(q) \cup V(g_1) \cup ... \cup V(g_n)$, to be added/deleted into/from the initial user query graph q. However, computing both *mcs* and the distance between graphs are NP-hard problems ([9], [8], [13] for survey). Therefore, approximate solutions are required. In order to minimize the distances between the new query graph q^* and graphs g_i in X, we present a novel approximate solution with polynomial time complexity while maintaining high quality. Our *why-not* question answering approach for similarity matching in graph databases compares the query graph and graph set X based on their sub-units, and then finds an approximate query graph q^* after approximating the *mcs* and the distances between the query graph q and graphs $g_i \in X$.

Our main contributions are summarized as follows. We propose a novel two-step solution approach consisting of candidate generation and selection phases. Before starting the solution approach, we first define the problem formally and establish the search space of the new query graph. We then establish the approximate upper bound of the distances between the query graph and the graphs in the answer set, which gives us an objective function that we need to minimize in the new query graph. We then find an starting point for the new query graph, which is an approximate maximum common subgraph (*amcs*) between the query graph and graphs in the new answer set. Finally, we present our two-phase algorithm for constructing the approximate new query graph. In the first phase, we generate a set of candidate edges that need to be added/deleted into/from the initial query graph. In the second phase, we propose three algorithms to select a subset of the candidate edges for minimizing the objective function established for the new query graph. We also conduct extensive experiments on two real datasets to demonstrate the effectiveness and efficiency of our approach.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 provides the preliminaries. Section 4 presents the problem. Section 5 discusses our approach. Section 6 illustrates our two-step algorithm. Section 7 demonstrates the effectiveness of our approach and presents the performance results. Finally, Section 8 concludes the paper.

2 Related Work

Why-Not questions in Databases: This work is greatly motivated by the previous studies [14], [2], [3], [4] and [5] of answering why-not questions in different data and query settings. In [14], Huang et al propose to modify the original tuple values in the database so that why-not (missing) tuples become part of the SPJ (Select-Project-Join) query output. In [2], Chapman et al. propose to identify the culprit operator(s) that filters out the why-not (missing) tuple(s) from the query output. As a next step, Tran and Chan [3] answer why-not questions for SPJA (Select-Project-Join-Aggregation) queries through query refinement where they collect why-not (missing) tuples as feedback from the user. In [4], He et al. propose an approach to answer why-not questions on top-k queries through the modification of both k and weightings in user preference queries. In [5], Islam et al. propose an approach for answering *why-not* questions in reverse skyline queries to start an automatic negotiation between customer and product of the company. In [15], Islam et al. presents an approach of modifying the initial SPJ query by exploiting the skyline operator if both *why* and *why-not* tuples are provided by the user. Here, we address the problem of answering why-not questions for graphs via modifying the initial query into a new query so that the new query includes the missing graphs in the answer set as well as minimizes the distance with the set. To the best of our knowledge, this problem is not addressed in the existing literature.

Graph Matching and Searching: Graph matching deals with the problem of finding matching or similarity score of two graphs. There are two broad categories in this direction: (a) exact graph matching and (b) approximate graph matching. For exact graph matching, most of the algorithms use backtracking (subgraph and graph isomorphism [16], [17]) and focus on maximum common node induced subgraph, or maximum clique approach [18], [19]. For approximate graph matching, there are three categories: (a) propagation based method [20] (b) spectral based method [21] and (c) optimization based method [22]. The work in [23] develops a feature-based filtering algorithm (as the pairwise substructure similarity computation is very expensive) for substructure similarity search in graph databases. The work in [24] investigates fast subgraph matching technique which can relax both structural and label matching constraints of subgraph isomorphism and graph similarity measures for query answering. The work in [13] finds matching of two large graphs in terms of the possibly maximum number of matched edges. A very recent work in [25] proposes a two-step process to (a) identify the structure in the data repository that the user is indicating in the query (a query is a sample from the desired result set) and (b) find the remaining structures based on the

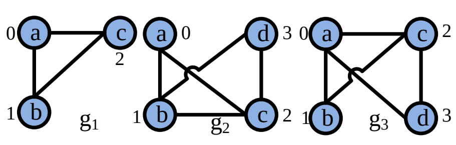


Fig. 1. A sample set of graphs

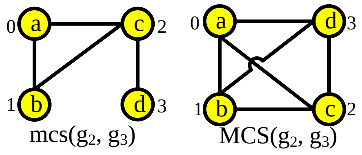


Fig. 2. The mcs and MCS of g_2 and g_3

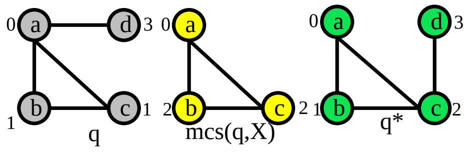


Fig. 3. The initial query, $mcs(q, X)$ and the modified query

structure that has been identified in the first step. In our work, we aim to maximize the matching between the query graph q and a set of expected answer graphs X by adding/deleting certain edges to/from q as well as minimize the distance (number of mismatched edges) between the new query graph q^* and the expected answer graph set X. In our problem setting, we also assume that the *why-not* (missing) graphs provided by the user are not too distant from the initial query graph q. That is, our problem falls into explaining missing answer graphs via graph query refinement category, rather than querying graph databases by example graphs as in [25].

Median Graph: Our problem setting is closely related to the problem of computing the median graph. Given a set of graphs, $X = \{g_1, g_2, ..., g_n\}$, the generalized median graph is a graph g that minimizes the sum of distances (SOD) to all the graphs in X [26]. The median graph is used as a representative of a set of graphs [26], [27], [28]. The computation of the generalized median graph is NP-hard and therefore, approximate solutions are preferred [26], [29]. An alternative to the generalized median graph, which is computationally less challenging, is the set median graph [30] (if g is a member of X, then g is called the set median graph). One may argue that our problem can be solved by the generalized median graphs. Median graph minimizes the SOD, not the distance with the set. Two solutions may have the same SOD with different distances. Therefore, median graph does not work for our problem (experiments suggest this too). Our intention is minimizing the distance between the query graph q and the graph set X, which is different from computing the median graph. However, if there are more than one modified queries with the same distance, we accept the one with minimized SOD to break the tie.

3 Preliminaries

Without loss of generality, we consider simple graphs in this paper. A simple graph, also called a strict graph, is an unweighted, undirected graph containing no self-loops or multiple edges. Given a set of vertex-labels, $\sum_V$, a simple graph is denoted by $g = (V, E, l)$ where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, l is a vertex labeling function, $l : V \to \sum_V$. For each vertex $u \in V$, $l(u)$ denotes the label of u. We denote the vertex set and the edge set of g by $V(g)$ and $E(g)$, respectively. $|V(g)|$ and $|E(g)|$ represent the numbers of vertices and edges in graph g, respectively.

Definition 1: Given two graphs $g_1 = (V_1, E_1, l_1)$ and $g_2 = (V_2, E_2, l_2)$, g_1 is subgraph isomorphic to g_2, denoted by $g_1 \subseteq g_2$, if there is an injective function $f : V(g_1) \to V(g_2)$ such that (a) $\forall u, v \in V_1$ and $u \neq v$, we have $f(u) \neq f(v)$. (b) $\forall v \in V_1$, we have $f(v) \in V_2$ and $l_1(v) = l_2(f(v))$. (c) $\forall (u, v) \in E(g_1)$, $(f(u), f(v)) \in E(g_2)$. We also say g_2 is a supergraph of g_1. If g_1 and g_2 are subgraph isomorphic to each other, we say g_1 and g_2 are isomorphic ($g_1 \equiv g_2$).

Example 1: Consider the graphs given in Fig. 1. The graph g_1 is subgraph isomorphic to g_2 and g_3 and both g_2 and g_3 are supergraphs for g_1, i.e., $g_1 \subseteq g_2$ and $g_1 \subseteq g_3$.

Definition 2: A graph g is a common subgraph of g_1 and g_2, if g is subgraph isomorphic to both g_1 and g_2. A graph g is a common supergraph of g_1 and g_2, if both g_1 and g_2 is subgraph isomorphic to g.

Definition 3: Graph g is a maximum common subgraph (mcs) of two graphs g_1 and g_2, if g is a common subgraph of g_1 and g_2, and there is no other common subgraph g' of g_1 and g_2, such that $|E(g')| > |E(g)|$.

Definition 4: Graph g is a minimum common supergraph (MCS) of two graphs g_1 and g_2, if both g_1 and g_2 is subgraph isomorphic to g, and there is no other common supergraph g', such that $|E(g)| > |E(g')|$.

Example 2: Consider the graphs given in Fig. 1. The mcs and MCS of g_2 and g_3 are shown in Fig. 2.

Definition 5: Given two graphs g_1 and g_2, the graph distance (GD) between g_1 and g_2, denoted by $\lambda(g_1, g_2)$, is defined as follows:

$$\lambda(g_1, g_2) = |E(g_1)| + |E(g_2)| - 2 \times |E(mcs(g_1, g_2))| \quad (1)$$

Example 3: Consider the graphs given in Fig. 1. We see that the graph distance between g_1 and g_2 is 2 ($\lambda(g_1, g_2) = |E(g_1)| + |E(g_2)| - 2 \times |E(mcs(g_1, g_2))| = 3 + 5 - 2 \times 3$). Also, the distance between g_1 and g_3 is 2 ($\lambda(g_1, g_3) = |E(g_1)| + |E(g_3)| - 2 \times |E(mcs(g_1, g_3))| = 3 + 5 - 2 \times 3$).

The justification of using GD computed by Eq. 1 in similar structure matching (e.g., chemical structure databases) can be found in [8]. It is easy to verify that the distance measure given in Eq. 1 fulfills the following properties of a metric: (a) if g_1 and g_2 are isomorphic, then $\lambda(g_1, g_2) = 0$ and (b) $\lambda(g_1, g_2)$=$\lambda(g_2, g_1)$.

Definition 6: Given a set of graphs $X = \{g_1, g_2, ..., g_n\}$, the distance bound between the query graph q and the graph set X, denoted by $\lambda_m(q, X)$, is defined as follows:

$$\lambda_m(q, X) = max\{\lambda(q, g_i)|g_i \in X\} \quad (2)$$

We use $\lambda_m(q, X)$ and λ_m interchangeably in the paper.

Example 4: Consider the graphs given in Fig. 1 as X and the query graph q given in Fig. 3. Then, the distance bound between q and X is $\lambda_m = max\{1, 3, 1\} = 3$.

Definition 7: Given a graph database consisting of N graphs as $D = \{g_1, g_2, ..., g_N\}$ and a query graph q, in similar graph searching the system retrieves either the

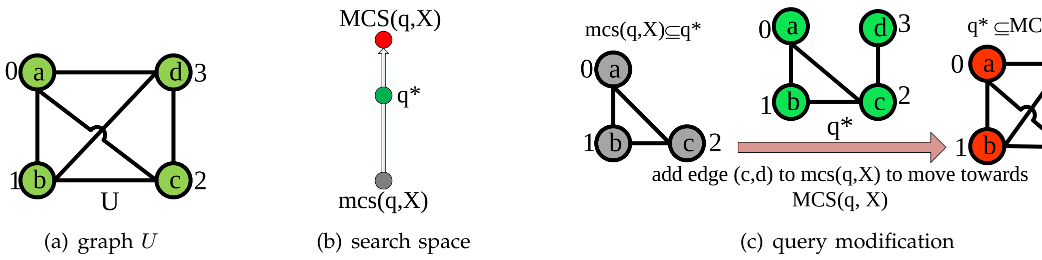


Fig. 4. The largest possible graph U, the search space and construction of the new query graph q^*

top-k graphs from D with the k smallest graph distances from q (e.g., $TOP_k(q, D)$ [8]) or graph set X whose distance bound with the query graph q are less than a given threshold δ, i.e., $\lambda_m(q, X) \leq \delta$ [9].

Searching similar graphs in database D for q requires computing the graph distance given in Eq. 1. But, computing exact graph distance between graphs is an NP-hard problem [9], [8]. Therefore, efficient ways of searching similar graphs in databases estimate the lower bound of graph distances between the query graph q and graphs in D to filter the non-promising graphs in D before computing the exact graph distances and thereby, can save lots of computational time ([8], [9] for survey).

Example 5: Consider the graphs given in Fig. 1 are the only graphs in D. Now, assume that a user wants to retrieve top-2 similar graphs from database D for the query graph q given in Fig. 3 (or graph set $X_1 = \{g_i\} \subset D$ with $\lambda_m(q, X_1) \leq 1$). The distance between q and the graphs in D are $\lambda(q, g_1) = 1$, $\lambda(q, g_2) = 3$ and $\lambda(q, g_3) = 1$. So, $TOP_2(q, D)$ or X_1 is $\{g_1, g_3\}$, where $\lambda(q, X_1)$ is $\{1, 1\}$ and $\lambda_m(q, X_1)$ is $max\{1, 1\} = 1$.

4 Problem Statement

Consider the graphs given in Fig. 1 and the query graph q given in Fig. 3. Assume that the underlying system retrieves only g_1 and g_3 while querying for q in a graph database D (see Example 5). A user may wonder *why* the system did *not* retrieve g_2, i.e., g_2 is expected in the answer set. Is g_2 too dissimilar to q? The user can try other queries for retrieving g_2. However, this requires a number of trials which is burdensome for the user. It may also happen that the user never finds the best q. Another problem is that the inclusion of missing graphs in the answer set of the random new query may violate the notion of similarity matching and also may not retain the previous results. Therefore, this may not be helpful. She may then ask what changes in her query could possibly include the missing graphs in the new answer set. We argue that the user can provide feedback on missing graphs and the system can then modify the query q for her to address the above. A minimum requirement of this kind of modification is that the new query q^* must minimize the distance bound with the answer set including the missing graphs to preserve the notion of similarity matching. Therefore, we formally define the problem as given below.

Definition 8: Given a set of graphs $X = \{g_1, g_2, ..., g_k, g_{k+1}, g_{k+2}, ..., g_n\}$, where $X_1 = \{g_1, g_2, ..., g_k\}$ is retrieved by the system in response to the user query graph q and $X_2 = \{g_{k+1}, g_{k+2}, ..., g_n\}$ is the set of missing graphs in the answer set, we want to modify the initial query graph q into a new query graph q^* such that q^* meets the following:

$$q^* = \underset{q' \in U}{\text{argmin}}\ \lambda_m(q', X) \tag{3}$$

where $q' \subseteq U$, $V(U) = V(g_1) \cup ... \cup V(g_n) \cup V(q)$ and $E(U) = V(U) \times V(U)$. The universal graph U represents the largest graph consisting of all vertices from the query graph q and graphs g_i in X, as shown in Fig. 4(a). The minimal vertex set for U can only be found by finding the optimal vertex to vertex mapping between $V(q)$ and $V(g_i) \in X$ (vertex mapping is explained in Section 5). Each $g_i \in X$ and q is subgraph isomorphic to U.

We define the quality of modifying q into q^* as follows:

$$\theta(q, q^*) = 1 - \frac{\lambda_m(q^*, X)}{\lambda_m(q, X)},\ \lambda_m(q^*, X) \leq \lambda_m(q, X) \tag{4}$$

The greater the value of θ the better the modified query q^* is ($0 \leq \theta \leq 1$). Any q', not necessarily be the best one (i.e., q^*), is a feasible solution if $\lambda_m(q', X) \leq \lambda_m(q, X)$. Eq. 4 emphasizes on matching the expected answer set X in the new query q^* better than the original query q. In connection with this, we assume the following: if a graph $g_1 \in X$ is dominated by a graph g_2 in terms of distance to q^*, i.e., $\lambda(q^*, g_2) \leq \lambda(q^*, g_1)$, then g_2 is an *implicit why-not* (missing) graph in relation to the *explicit why-not* feedback g_1 provided by the user. The value of θ remains the same for the inclusion of these implicit *why-not* graphs in the answer set of q^*.

Definition 9: The *mcs* of q and the graph set X, denoted by $mcs(q, X)$, is the graph g which satisfies the followings: (a) g is subgraph isomorphic to q and each $g_i \in X$, i.e., $g \subseteq q$ and $g \subseteq g_i$, $\forall g_i \in X$ and (b) there exists no other graph g' such that g' is also subgraph isomorphic to q and each $g_i \in X$ and $|E(g')| > |E(g)|$.

Lemma 1: The modified query graph q^* retaining the $mcs(q, X)$ offers the best quality.

Example 6: Consider the graphs given in Fig. 1 as X, the query graph q and the $mcs(q, X)$ in Fig. 3. The modified query q^* is also shown in Fig. 3 which retains the $mcs(q, X)$. We compute $\lambda_m(q^*, X)$ as 1 and the quality θ is 0.67 ($1 - \frac{\lambda_m(q^*, X)}{\lambda_m(q, X)} = 1 - \frac{max\{\lambda(q^*, g_1), \lambda(q^*, g_2), \lambda(q^*, g_3)\}}{max\{\lambda(q, g_1), \lambda(q, g_2), \lambda(q, g_3)\}} = 1 - \frac{max\{1,1,1\}}{max\{1,3,1\}} = 1 - \frac{1}{3}$). It is easy to verify that any other candidate query q' that does not retain $mcs(q, X)$ must offer $\theta(q, q') < 0.67$ as $\lambda_m(q', X) \geq \lambda_m(q^*, X) + 1$.

However, computing the modified query graph q^* requires not only computing the $mcs(q, X)$ but also adding/deleting certain edges into/from q to minimize $\lambda_m(q^*, X)$. We argue that we can not add/delete edges into/from q arbitrarily, but the candidate edges must be in $E(g_i)$ and/or in $E(q)$.

Definition 10: The *MCS* of q and the graph set X, denoted by $MCS(q, X)$, is the graph g which satisfies the followings: (a) g is the supergraph of q and each $g_i \in X$, i.e., $q \subseteq g$ and $g_i \subseteq g$, $\forall g_i \in X$ and (b) there exists no other graph g' such that g' is also the supergraph of q and each $g_i \in X$ and $|E(g)| > |E(g')|$.

Lemma 2: The new query q^* is subgraph isomorphic to $MCS(q, X)$ and $mcs(q, X)$ is subgraph isomorphic to q^*. That is, $mcs(q, X) \subseteq q^* \subseteq MCS(q, X)$.

The search space of the new query graph q^* is depicted in Fig. 4 (b) (imaginative view). The directed vertical line marked with $mcs(q, X)$ and $MCS(q, X)$ signifies the true search space for optimal q^*, i.e., the new query q^* lies in between $mcs(q, X)$ and $MCS(q, X)$ ($mcs(q, X) \subseteq q^* \subseteq MCS(q, X) \subseteq U$). Therefore, to find the new query, we start from $mcs(q, X)$ (Lemma 1) and move towards $MCS(q, X)$ (Lemma 2) via candidate edges generation and selection (edges to be added and/deleted from the intial query graph q) as shown in Fig. 4(c) (the two-phase algorithm is described in Section 6).

Hardness of the Problem: From Lemma 1, we see that computing the new query graph q^* with best quality requires computing the *mcs* of the query graph q and the graph set X. However, computing *mcs* between graphs is an NP-hard problem [8]. Computing the optimal q^* also requires computing the graph distances, $\lambda(q, g_i)$, $\forall g_i \in X$. Again, the computation of exact graph distance is an NP-hard problem [8], [9]. Therefore, answering *why-not* questions in similar graph matching with the objective function given in Eq. 3 is an NP-hard problem.

5 Our Approach

As the computation of optimum query graph q^* is an NP-hard problem, we propose to compute an *approximated maximum common subgraph* of the query graph q and the graph set X, denoted by $amcs(q, X)$ and also an approximated upper bound of $\lambda(q, g_i)$, denoted by $\overline{\lambda}(q, g_i)$, to compute an approximation for q^*.

The computation of $amcs(q, X)$ and $\overline{\lambda}_m(q, X)$ requires computing the best mapping f between the vertices of the query graph q and vertices of the graphs $g_i \in X$, which maximizes the number of matched edges in $amcs(q, X)$ as many as possible. To find the best possible mapping, we adapt the idea proposed in [9] to decompose a graph into multiple sub-units like *star*. A *star* is a labeled, single-level and rooted tree which is represented by a 3-tuple $s = (r, L, l)$, where r is the root, L is the set of leaves and l is a labeling function. For each $v \in g$, we construct a star $s = (v, L, l)$, i.e., the graph g is decomposed into a multiset of $|V(g)|$ stars. For example, the sample graph g_1 and the query graph q are decomposed into two star representations: $S(g_1)$ and $S(q)$ as shown in Fig. 5. Now, we define the following.

Definition 11: Given two stars s_1 and s_2, the star similarity (SS) between them is computed as follows:

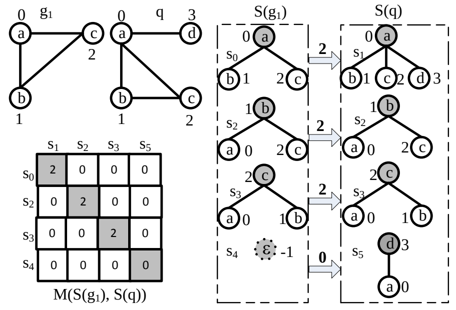


Fig. 5. Mapping between $S(g_1)$ and $S(q)$ based on SS

$$\Lambda(s_1, s_2) = \begin{cases} |\Psi_{L_1} \cap \Psi_{L_2}| & \text{if } l(r_1) = l(r_2) \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where, Ψ_L is the multiset of vertex labels in L. The SS defined in Eq. 5 is different from the *Star Edit Distance* (SED) defined in [9]. In SS, we emphasize on structural (edge) similarity where we assume vertex insertions/deletions are implicitly accomplished with their incident edge insertions/deletions. This is because we want to maximize the number of matched edges in $amcs(q, X)$. Distance optimality does not necessarily mean matched edge optimality. Now, the computation of mapping f between the vertices of q and vertices of $g_i \in X$ is equivalent to finding an optimal mapping between the star representations of q and $g_i \in X$ [9]. The optimal mapping $P : S(q) \rightarrow S(g_i)$ between stars of $S(q)$ and $S(g_i)$ based on SS is defined as follows:

$$\min_P \sum_{s_q \in S(q)} \Lambda(s_q, P(s_q)) \tag{6}$$

To compute the optimal mapping P, we first construct a weighted (square) matrix for each pair of stars from q and $g_i \in X$ and then apply the Hungarian algorithm (HA) [31] to get the optimal solution in cubic time. The weight between two stars is set to SS. If the graphs are of different size, ε vertex is inserted for normalization.

Example 7: The SS between $s_0(g_1)$ and $s_1(q)$ is $\Lambda(s_0, s_1) = |\Psi_{L_0} \cap \Psi_{L_1}| = 2$, as $l(r_0) = l(r_1)$. The SS between $s_0(g_1)$ and $s_2(q)$ is $\Lambda(s_0, s_2) = 0$ as $l(r_0) \neq l(r_2)$ as shown in Fig. 5. The bottom left matrix $M(S(g_1), S(q))$ in Fig. 5 is the weight matrix between star sets $S(g_1)$ and $S(q)$ based on SS. Cells in gray denote the optimal matching between $S(g_1)$ and $S(q)$, and the mappings are marked with arrows on the right in Fig. 5.

Once we compute the optimal mapping between the two star representations of q and $g_i \in X$, we can find the mapping between the vertices of q and g_i easily. The roots of the mapped stars of $S(q)$ and $S(g_i)$ map to each other and thereby represents the mapping between vertices of q and $g_i \in X$, i.e., $f(r(s_q)) \leftarrow r(P(s_q))$. Here, f is a bijective vertex to vertex mapping function, i.e., if $f(u) = v$, then $f^{-1}(v) = u$. For example, the vertex 0 with label 'a' in graph g_1 mapped to the vertex 0 with label 'a' in the query graph as their corresponding stars $s_0(g_1)$ and $s_1(q)$ mapped by P.

The computation of $amcs$ between the query graph q and a graph $g_i \in X$, denoted by $amcs(q, g_i)$, becomes straightforward. An edge $(u, v) \in E(q)$ is added to the $amcs(q, g_i)$ if the follwowing holds: (a) $l_q(u) = l_{g_i}(f(u))$; (b) $l_q(v) = l_{g_i}(f(v))$ and (c) $(f(u), f(v)) \in E(g_i)$.

Lemma 3: The $amcs(q, g_i)$ is suboptimal to $mcs(q, g_i)$, i.e., $|E(amcs(q, g_i))| \leq |E(mcs(q, g_i))|$.

The $amcs(q, g_i)$ can be used to approximate the graph distance between q and g_i given in Eq. 1 as follows:

$$\overline{\lambda}(q, g_i) = |E(q)| + |E(g_i)| - 2 \times |E(amcs(q, g_i))| \quad (7)$$

Lemma 4: The graph distance computed by Eq. 7 is an upper bound of $\lambda(q, g_i)$, i.e., $\overline{\lambda}(q, g_i) \geq \lambda(q, g_i)$.

Example 8: Consider the graphs given in Fig. 1 as X and the query graph q in Fig. 3. The vertex to vertex mapping for q and g_1 is shown in Fig. 5 and other graphs are shown in Fig. 6. We compute the upper bound of GD of q with each graph g_i in X as follows: $\overline{\lambda}(q, g_1) = |E(q)| + |E(g_1)| - 2 \times |E(amcs(q, g_1))| = 4 + 3 - 2 \times 3 = 1$, $\overline{\lambda}(q, g_2) = |E(q)| + |E(g_2)| - 2 \times |E(amcs(q, g_2))| = 4 + 5 - 2 \times 3 = 3$ and $\overline{\lambda}(q, g_3) = |E(q)| + |E(g_3)| - 2 \times |E(amcs(q, g_3))| = 4 + 5 - 2 \times 4 = 1$. Therefore, the corresponding distance bound of q and X is $\overline{\lambda}_m(q, X) = max\{1, 3, 1\} = 3 \geq \lambda_m(q, X) = 3$.

Once we compute the $amcs$ between the query q and each graph $g_i \in X$, we compute the $amcs$ of q and the graph set X by following the equation given as follows:

$$amcs(q, X) = \bigcap_{g_i \in X} amcs(q, g_i) \quad (8)$$

The above (Eq. 8) is realized as follows. Firstly, we compute the $amcs$ between the query graph q and the first graph g_i in X, i.e., $amcs(q, g_i)$. Then, we compute the $amcs$ between the next graph g_j in X and the previously computed $amcs(q, g_i)$. We continue this process until we finish all graphs in X. The final $amcs$ becomes the $amcs(q, X)$. In our approach, we retain the $amcs(q, X)$ in the new query graph (Lemma 1).

In connection with the avobe approximation, we redefine the objective function given in Eq. 3 as follows:

$$\hat{q} = \underset{q' \in U}{\operatorname{argmin}}\ \overline{\lambda}_m(q', X) \quad (9)$$

where $\overline{\lambda}_m(q', X) = max\{\overline{\lambda}(q', g_i) | g_i \in X\}$, $q' \subseteq U$ and $\hat{q}$ is the approximation for q^*. The corresponding quality of the approximated new query $\hat{q}$ is quantified as follows:

$$\hat{\theta}(q, \hat{q}) = 1 - \frac{\overline{\lambda}_m(\hat{q}, X)}{\overline{\lambda}_m(q, X)}, \overline{\lambda}_m(\hat{q}, X) \leq \overline{\lambda}_m(q, X) \quad (10)$$

The quality of the new query $\hat{q}$ returned by an approximation algorithm depends on the quality of $amcs(q, X)$ and $\overline{\lambda}(q, g_i)$ computed. Here, we adapt the Hungarian Algorithm [31] based bipartite graph matching techniques proposed in [8], [9] for computing both $\overline{\lambda}_m(q, X)$ and $amcs(q, X)$. In the new query $\hat{q}$, we retain $amcs(q, X)$ and then, we move towards $MCS(q, X)$ via candidate edge generation (mismatched edges between q and X) and selection which is explained in Section 6. Here, we select edges from q and $g_i \in X$ to derive the new query $\hat{q}$ in a way that keeps $\hat{q}$ as close as possible to both q and $g_i \in X$. We assume that the graphs in X_2 are not too distant. For distant missing (i.e., why-not) objects, we propose to accept those new queries that minimize the upper bound of graph distances better than the original one and let the user to decide on this. That is, our problem setting transforms to querying graphs by example(s) for far distant missing objects.

6 The Algorithm

This section describes our two-phase algorithm for constructing the new query graph $\hat{q}$, which consists of a candidate generation phase and a candidate selection phase. In the candidate generation phase, we generate a set of candidate edges to be added/deleted into/from the query graph q. In the candidate selection phase, we select a subset of the candidates generated in the first phase to minimize the distance bound between the new query $\hat{q}$ and the graph set X.

6.1 Candidate Generation

To find the candidate edges to be added/deleted into/from the query graph q, we first decompose the query graph q and the graphs g_i in X into their constituent stars. Then, we find the matching stars of the query graph q and the graphs g_i in X by running HA (pairwise matching between the stars of q and $g_i \in X$) and the corresponding vertex to vertex mapping f (we have already described it for computing $amcs(q, X)$ and $\overline{\lambda}_m(q, X)$ in Section 5, we reuse them here). Then, we find the edges between q and g_i which are not common to them (common edges between q and X are part of $amcs(q, X)$ and are retained in the new query $\hat{q}$).

If a mismatched edge is found, we generate a candidate operation i.e., *add*/*del* for it. If the mismatched edge belongs to q, we generate a candidate *del* operation for it, $o = del(u, v)$ (lines 4-7 in Algorithm 1). Otherwise, we generate a candidate *add* operation for it, $o = add(u, v)$ lines 8-17 in Algorithm 1). The graphs g_i that will have effect on their $\overline{\lambda}(q, g_i)$ are saved in the corresponding operation's $gList$. Each candidate operation o is associated with a cost vector c of size $|X|$, an entry for each $g_i \in X$ in it. Each entry of c can be either -1 or $+1$ if it decreases or increases the value of $\lambda(q, g_i)$, respectively. The vector c is initialized with $1^{|X|}$, which is updated according to the graphs g_i found in the corresponding operation's $gList$ (lines 18-21 in Algorithm 1).

While generating candidate edges to be added/deleted into/from the query graph q, there is a possibility of generating duplicate candidate edges in the candidate edge generation phase. The duplicate edges are detected by maintaining the vertex number and its label from the query graph q and its corresponding matched vertices in $g_i \in X$ (e.g., formation of candidate edge operation in line 6). That is, query graph vertices are treated as the reference for each graph g_i in X. Duplicate edges are detected and are merged with the existing one as part of $addOperation$ functionality in lines 24-30 of Algorithm 1. We also need

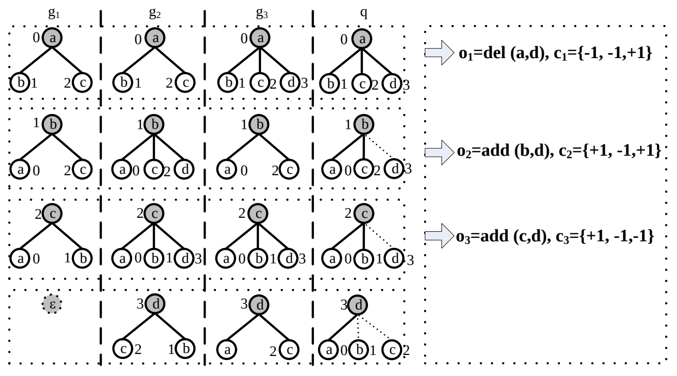
Algorithm 1: Candidate Generation

```
Input  : query graph ($q$), graph set ($X$), vertex-vertex mapping function ($f$)
Output: set of candidate edge operations ($O$)
1  begin
2      $O \leftarrow null$; // initialization
3      for $g_i \in X$ do
4          for $(u,v) \in E(q)$ do
5              if $l_q(u) \neq l_{g_i}(f(u))$ or $l_q(v) \neq l_{g_i}(f(v))$ or $(f(u), f(v)) \notin E(g_i)$ then
6                  $o.type \leftarrow del$; $o.edge \leftarrow (u,v)$;
                   $o.vLabels \leftarrow (l_q(u), l_q(v))$; $o.gList \leftarrow g_i$;
7              addOperation($O$, $o$); // save operation
8          for $(u,v) \in E(g_i)$ do
9              if $l_{g_i}(u) = l_q(f^{-1}(u))$ and $l_{g_i}(v) = l_q(f^{-1}(v))$ and $(f^{-1}(u), f^{-1}(v)) \notin E(q)$ then
10                 $o.type \leftarrow add$; $o.edge \leftarrow (f^{-1}(u), f^{-1}(v))$;
                   $o.vLabels \leftarrow (l_{g_i}(u), l_{g_i}(v))$; $o.gList \leftarrow g_i$;
11             else if $l_{g_i}(u) \neq l_q(f^{-1}(u))$ and $l_{g_i}(v) = l_q(f^{-1}(v))$ then
12                 $o.type \leftarrow add$; $o.edge \leftarrow (-1, f^{-1}(v))$;
                   $o.vLabels \leftarrow (l_{g_i}(u), l_{g_i}(v))$; $o.gList \leftarrow g_i$;
13             else if $l_{g_i}(u) = l_q(f^{-1}(u))$ and $l_{g_i}(v) \neq l_q(f^{-1}(v))$ then
14                 $o.type \leftarrow add$; $o.edge \leftarrow (f^{-1}(u), -1)$;
                   $o.vLabels \leftarrow (l_{g_i}(u), l_{g_i}(v))$; $o.gList \leftarrow g_i$;
15             else if $l_{g_i}(u) \neq l_q(f^{-1}(u))$ and $l_{g_i}(v) \neq l_q(f^{-1}(v))$ then
16                 $o.type \leftarrow add$; $o.edge \leftarrow (-1, -1)$;
                   $o.vLabels \leftarrow (l_{g_i}(u), l_{g_i}(v))$; $o.gList \leftarrow g_i$;
17             addOperation($O$, $o$); // save operation
18     for $o \in O$ do
19         $c \leftarrow 1^{|X|}$; // initialize vector $c$ with 1
20         for $g_i \in o.gList$ do
21             $c[g_i] \leftarrow -1$; // update $c[g_i]$ as $o$ decreases $\lambda(q, g_i)$

22 addOperation($O$, $o_1$)
23 begin
24     $flag \leftarrow false$;
25     for $o_2 \in O$ do
26         if $o_2.edge = o_1.edge$ and $o_2.vLabels = o_1.vLabels$ then
27             $o_2.gList \leftarrow o_2.gList \cup o_1.gList$; // update $o_2.gList$ by merging it with $o_1.gList$
28             $flag \leftarrow true$; break; // duplicate entry
29     if $flag = false$ then
30         $O \leftarrow O \cup o_1$; // an entry for this new operation is created
```

to insert hypothetical vertices in q if the answer graph vertices do not already exist in q (numbered by -1 in lines 11-16 of Algorithm 1) to serve the above.

Example 9: Consider the graphs given in Fig. 1 as X and the query graph given in Fig. 3. The star representations of each graph $g_i \in X$ and the query graph q, and their corresponding matchings are shown in Fig. 6. The candidate operations generated by Algorithm 1 are also shown on the right side of Fig. 6. As an example, we can see in the first row of Fig. 6(a) that edge (a,d) does not appear in graphs g_1 and g_2 and therefore, we compute a del operation for it (steps 4 to 7 in Algorithm 1). However, if we delete this edge from q, it increases the distance with g_3 by 1 though it decreases the distances with both g_1 and g_2 by 1, which becomes the cost for deleting this edge from q (computed in steps 18 to 21 in Algorithm 1). Consider the second row of Fig. 6(a). We see that edge (b,d) appears in graph g_2, but not in q. Therefore, we compute an add operation for it (steps 8


g1
g2
g3
q
o1=del (a,d), c1={-1, -1,+1}
o2=add (b,d), c2={+1, -1,+1}
o3=add (c,d), c3={+1, -1,-1}
ε

(a) Matching stars of q and $g_i \in$ X (b) Candidate generation

Fig. 6. Candidate generation for modifying the query q given in Fig. 3 and graphs given in Fig. 1

to 17 in Algorithm 1). The cost of adding (b,d) into the query graph q is $\{+1,-1,+1\}$. The first and last entries of c_2 is +1 as (b,d) does not appear in g_1 and g_3 (i.e., addition of (b,d) into q increases $\lambda(q,g_1)$ and $\lambda(q,g_3)$ by 1). Similarly, we compute the add operation o_3(add (c,d), $c_3 = \{+1,-1,-1\}$) which is shown in Fig. 6.

Complexity Analysis. The two inner *for* loops in line 4 and line 8, respectively, execute sequentially for each graph $g_i \in X$ iterated in the outer *for* loop in line 3. Assume that the average number of edges in a graph $g_i \in X$ is $|E(g)|$. Also, assume that the $addOperation$ function and if statements take constant time to execute. Therefore, the complexity of lines 3-17 becomes $\mathcal{O}(|X| \times (|E(q)| + |E(g)|))$. The maximum number of operations for the *for* loop in line 18 can be $|E(q)| + |X| \times |E(g)|$. The inner for loop of line 18 in line 20 can run $|X|$ times, for which the complexity of line 18-21 becomes $\mathcal{O}(|X| \times |E(q)| + |X|^2 \times |E(g)|)$ Therefore, the overall worst-case complexity of our candidate edge generation becomes $\mathcal{O}(|X| \times (|E(q)| + (1+|X|) \times |E(g)|))$.

6.2 Candidate Selection

In this phase, we select a subset of the candidates generated in the first phase to minimize our objective function given in Eq. 9. However, the candidate selection phase exposes as combinatorial optimization problem, i.e., selecting a subset out of $2^{|O|}$ subsets. Another problem is that the solution may not necessarily be unique. To solve this, we accept the query $\hat{q}$ with the *minimum sum of distances* (SOD) with X, which is defined as follows:

$$SOD(\hat{q}, X) = \sum_{g_i \in X} \overline{\lambda}(\hat{q}, g_i) \tag{11}$$

The intuition of using SOD for breaking the tie is that two $\hat{q}$ with the same $\overline{\lambda}_m$ may have different $SODs$ with X. The minimized SOD makes the refined query $\hat{q}$ representative of the graph set X [26]. Now, we present three different candidate selection algorithms (with different merits) for selecting a subset of candidates from O:

1) Backtracking with Pruning (BP);
2) Greedy Search (GS) and
3) Genetic Programming (GP) based selection.

Algorithm 2: BP based Candidate Selection

```
Input  : $\overline{\lambda}(q, X)$ and $O$: set of candidate edges
Output: $Z$: set of selected candidates
1  begin
2      $Z \leftarrow null$; $\overline{\lambda}_{cur} \leftarrow \overline{\lambda}(q, X)$; // initialization
3      $\overline{\lambda}_m \leftarrow max(\overline{\lambda}_{cur})$; $SOD \leftarrow sum(\overline{\lambda}_{cur})$;
4      sort $(O)$; // sort entries in descending order
5      construct cost matrix $C$; // a two dimensional array
6      $NC \leftarrow \{0\}^{|O|+1 \times |X|}$; // a two dimensional array
7      for $|O| \geq l \geq 1$ do
8          for $0 \leq j \leq |C_l|$ do
9              $NC_l[j] \leftarrow NC_{l+1}[j] + (C_l[j] = -1?1 : 0)$;
10     $SEL \leftarrow \{0\}^{|O|}$; // one dimensional array
11     dfs$(0, NC, O, \overline{\lambda}_m, SOD, \overline{\lambda}_{cur})$; // BP search
12     for $1 \leq j \leq |O|$ do
13         if $SEL[j] = 1$ then
14             add $o_j$ into $Z$;
15 Procedure dfs$(l, NC, O, \overline{\lambda}_{m_1}, SOD_1, \overline{\lambda}_{cur}, SEL_1)$
16 begin
17     if $p = |O|$ then
18         if $\overline{\lambda}_{m_1} < \overline{\lambda}_m$ or ($\overline{\lambda}_{m_1} = \overline{\lambda}_m$ and $SOD_1 < SOD$) then
19             $\overline{\lambda}_m \leftarrow \overline{\lambda}_{m_1}$; $SEL \leftarrow SEL_1$; $SOD \leftarrow sum(\overline{\lambda}_{cur})$;
20         return; // all combinations finished
21     for $1 \leq j \leq |\overline{\lambda}_{cur}|$ do
22         if $\overline{\lambda}_{cur}[j] - NC_l[j] \geq \overline{\lambda}_{m_1}$ then
23             return; // pruning next nodes
24     $\overline{\lambda}_{cur} \leftarrow \overline{\lambda}_{cur} + C_l$; $SEL_1[l] = 1$;
25     dfs$(l + 1, NC, O, max(\overline{\lambda}_{cur}), sum(\overline{\lambda}_{cur}), \overline{\lambda}_{cur})$;
26     $\overline{\lambda}_{cur} \leftarrow \overline{\lambda}_{cur} - C_l$; $SEL_1[l] = 0$;
27     dfs$(l + 1, NC, O, \overline{\lambda}_{m_1}, SOD_1, \overline{\lambda}_{cur})$;
```

6.2.1 *Backtracking with Pruning (BP)*

The BP is basically a binary decision tree-based candidate search algorithm. However, we do not enumerate all $2^{|O|}$ subsets completely while searching for the best candidate set in the tree. We preprocess the candidates in O to prune some non promising nodes and their descendants in the search tree, i.e., prune the subtree rooted at the current node to avoid some computations.

The steps are as follows: (1) sort the candidates $o \in O$ in descending order based on the number of -1's contained in their cost vectors c (line 4 in Algorithm 2); (2) construct a cost matrix C considering the sorted cost vectors c as rows (line 5 in Algorithm 2); (3) compute how many -1's are there for every cell ahead of it (vertically) in the matrix C including the cell itself as follows: (a) initialize NC with $0^{|O|+1 \times |X|}$ and (b) update NC as $NC_l[j] \leftarrow NC_{l+1}[j] + (C_l[j] = -1?1 : 0)$ (lines 6-9 in Algorithm 2), this information is used to decide whether a node will be expanded further in the search tree or not; and (4) start the search tree by creating a root node for the first candidate in sorted O and create two branches for it by labeling them as 1 (selected to modify q) and 0 (not selected to modify q), then in the next level of the search tree, create nodes for the next candidate and continue this process until all of the candidates in O are finished (lines 15-27 in Algorithm 2). While expanding the tree, if there is no possibility of reducing $\overline{\lambda}_m(\hat{q}, X)$ furthermore by the current node and all of it's descendants (i.e., if $\overline{\lambda}_{cur}[j] - NC_l[j] \geq \overline{\lambda}_m$), stop expanding the current node (lines 21-23 in Algorithm 2).

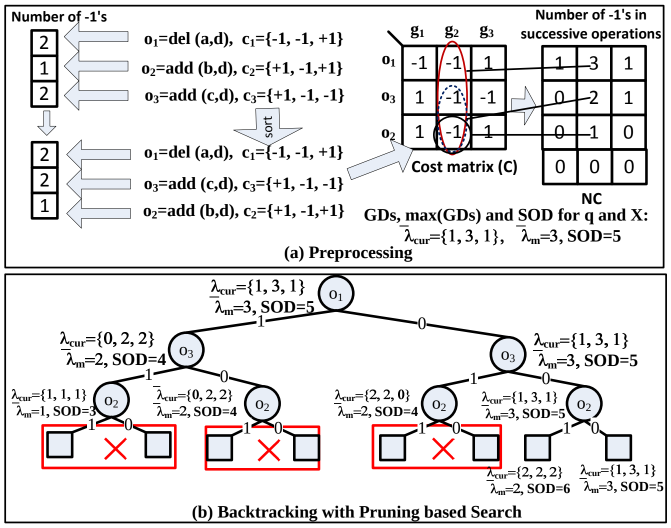


Fig. 7. BP based candidate selection

Every path in the tree represents a subset of candidate operations from O. But, we keep the one that minimizes $\overline{\lambda}_m(q, X)$ the most (lines 17-20 in Algorithm 2).

The worst-case time complexity of the above BP-based candidate selection is $\mathcal{O}(2^{|O|})$, i.e., similar to backtracking without pruning (BT). Therefore, BP is not time efficient if $|O|$ is large (i.e., search space size is $2^{|O|}$).

Example 10: From example 9, we get three candidate edge operations to modify our query graph q. To do a BP based candidate search, we first sort the candidates based on the number of -1's in them as shown in Fig. 7(a) i.e., $o_1 \geq o_3 \geq o_2$. Then, we construct the cost matrix C in which the rows represent the sorted candidates in O. Next, we compute NC, by intializing it with $0^{4 \times 3}$, where each cell contains the number of -1's ahead of it including itself in C (see Fig. 7(a)). After computing NC, we start our BP based candidate search starting with o_1 as root in the tree as shown in Fig. 7(b). Then, we create two branches with and without considering this candidate for modifying q. We then consider o_3 in the next level. We continue this process until we finish for o_2 or we prune the subtree for it as shown in Fig. 7(b). The leftmost path in the search tree gives us the best subset which consists of the operations o_1 and o_3. Finally, the quality of the modified query returned by BP becomes $\theta(q, \hat{q}) = 1 - \frac{\overline{\lambda}_m(\hat{q}, X)}{\overline{\lambda}_m(q, X)} = 1 - \frac{1}{3} = \frac{2}{3} = 0.67$.

6.2.2 *Greedy Search (GS)*

The main idea of GS-based candidate selection is to select a candidate from O that decreases the value of $\overline{\lambda}_m(\hat{q}, X)$ and/or $SOD(\hat{q}, X)$ most at the moment, and continue this process until O becomes empty or none of the remaining candidates in O can decrease the value of $\overline{\lambda}_m(\hat{q}, X)$ and/or $SOD(\hat{q}, X)$ furthermore. However, the selection of the next candidate depends on what candidates we have selected so far. Therefore, we need to re-rank every remaining candidates before next selection. The complexity of the GS based candidate selection is $\mathcal{O}(|O| \times |O|)$. The GS based candidate selection is very suitable if X contains very dissimilar graphs and the candidate size is very large, for which BP is inefficient. However, the GS based candidate selection does not guarantee to be as effective as BP in terms of quality of the query graph.

Example 11: Suppose that the new query $\hat{q}$ is initialized with q. We get $\overline{\lambda}(q, X) = \{1, 3, 1\}$ and $\overline{\lambda}_m = max\{1, 3, 1\} = 3$ from Example 8. Here, SOD is 5. According to the GS based candidate selection, firstly we select $o_3 \in O$ (i.e., add (c, d) into q), and we get $\overline{\lambda} = \{2, 2, 0\}$, $\overline{\lambda}_m = 2$ and SOD=4. As a next step, we select $o_1 \in O$ (i.e., delete (a, d) from q), and we get $\overline{\lambda} = \{1, 1, 1\}$, $\overline{\lambda}_m = 1$ and SOD=3. We find no more candidates in O that can improve $\overline{\lambda}_m$ and/or SOD furthermore. Finally, the quality of the new query returned by GS becomes $\theta(q, \hat{q}) = 1 - \frac{\overline{\lambda}_m(\hat{q},X)}{\overline{\lambda}_m(q,X)} = 1 - \frac{1}{3} = \frac{2}{3} = 0.67$, i.e., $\hat{q}$ matches X better than q.

6.2.3 Genetic Programming (GP)

The problem with BP is that it is very inefficient in terms of run-time complexity, but guaranteed to be effective in terms of quality (BP applies exhaustive search for searching suitable set of candidates). On the contrary, GS based candidate selection is computationally less expensive than BP. Unfortunately, GS does not guarantee to be as effective as BP in terms of quality. We propose a genetic programming (GP)-based candidate selection algorithm, which is a trade-off between the BP and GS based candidate edge selection algorithms.

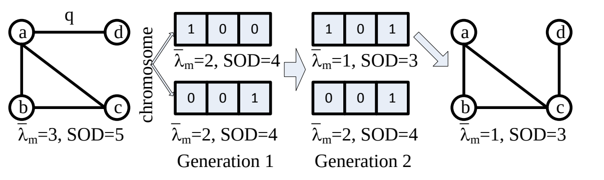
The basic steps of GP based candidate selection are as follows: A possible solution to the problem is encoded using *chromosomes*. Each chromosome has its own *fitness* which is computed by means of a *fitness function*. This *fitness* score is used to decide how good a chromosome is. Given an *initial population* of chromosomes, GP uses *genetic operators* to alter some of the chromosomes in the *population*, generating a *new population*. This process is iteratively repeated until one or more stop conditions are satisfied. For more information about genetic programming, interested readers are referred to [32], [26].

(a) Chromosome: A chromosome, ch, is a vector of size $|O|$ and each position in it is associated with one candidate in O. A position in ch can store either a value of '1' or '0' depending on whether the corresponding candidate in O is selected or not for modifying the query q. A *population G* consists of a set of chromosomes.

(b) Fitness function. The fitness $\mathcal{F}$ of each chromosome ch corresponding to the $\overline{\lambda}_m$ and SOD of the query $\hat{q}$ represented by the ch is given as follows.

$$\begin{aligned} \mathcal{F}_{\overline{\lambda}_m}(ch) &= max\{\overline{\lambda}(q, X) + \textstyle\sum_{j=1}^{|O|} ch[j] \times C_j\} \\ \mathcal{F}_{SOD}(ch) &= sum\{\overline{\lambda}(q, X) + \textstyle\sum_{j=1}^{|O|} ch[j] \times C_j\} \end{aligned} \qquad (12)$$

(c) Genetic operators. The classical operators of the genetic programming are *crossover* and *mutation*. The crossover operator simply interchanges an arbitrary position of two chromosomes (selected with a uniform probability) to form two offspring. Mutation is accomplished


q
a
b
c
d
$\overline{\lambda}_m$=3, SOD=5
chromosome
1 0 0
$\overline{\lambda}_m$=2, SOD=4
0 0 1
$\overline{\lambda}_m$=2, SOD=4
Generation 1
1 0 1
$\overline{\lambda}_m$=1, SOD=3
0 0 1
$\overline{\lambda}_m$=2, SOD=4
Generation 2
a
b
c
d
$\overline{\lambda}_m$=1, SOD=3

(a) Initial query graph (b) GP based search (c) Modified query graph

Fig. 8. GP based candidate selection

by flipping randomly a position in ch with a mutation probability. We apply genetic operators to create a new population. We also check every chromosome in order to validate its feasibility, which is realized by the fitness function given in Eq. 12.

(d) Population initialization. A population is initialized with a set of chromosomes, where each chromosome in the population represents a feasible solution for the new query graph. The performance of GP largely depends on the initial population. The size of the population pop_size is determined empirically.

(e) Termination condition. The evolution of population is continued until one of the following two conditions is met: (1) the maximum number of generations is reached and (2) the $\overline{\lambda}_m$ and SOD do not get improving.

An example for GP based candidate selection for graphs given in Fig. 1 and the initial query graph given in Fig. 3 is shown in Fig. 8, the evolution stops in this example after two generations. Once we stop GP search, we accept the chromosome with best fitness score as shown in Fig. 8. The quality of the modified query is $\theta(q, \hat{q}) = 1 - \frac{\overline{\lambda}_m(\hat{q},X)}{\overline{\lambda}_m(q,X)} = 1 - \frac{1}{3} = \frac{2}{3} = 0.67$. The GP based candidate selection is denoted by $GP_{x,y}$ in the rest of this paper, where x denotes the population size and y denotes the number of generations.

*The proofs of lemmas and the pseudocodes of GS and GP algorithms are given in the supplemental material.

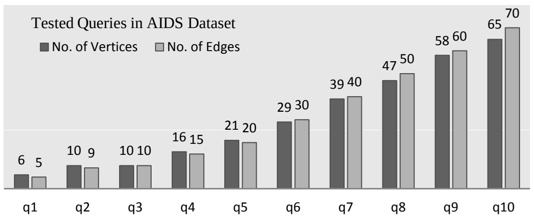
7 Experiments

This section demonstrates the effectiveness and efficiency of our approach for answering why-not questions in similar graph matching. All experiments are performed on a Windows PC with 2.99 GHz CPU and 3.49 GB main memory. All of our algorithms proposed in this paper are implemented in Java and Eclipse environment.

7.1 Setup

Datasets: We have used real AIDS Antiviral Screen chemical compound dataset, published by NCI [33], which consists of 42,687 chemical compounds, with an average of 46 vertices and 48 edges, and labeled with 63 unique vertex labels. AIDS has been widely used in many existing works including [8], [9]. The use of this dataset makes an excellent sense for our problem settings, i.e., asking *why-not* questions (missing targets) while querying the targets for the hypothetical drug. We have also used Linux dataset [9] (obtained by personal communication), which consists of 47,239 *program dependency graphs* (PDGs), with an average of 36 vertices and 38 edges, and 23315 unique vertex labels. PDG is a static

Parameter	Ranges
Datasets	AIDS and Linux
AIDS DB Size	42687 graphs
Linux DB Size	47239 graphs
Size of Answer Set	2−25
Size of Expected Set	1−25
Population Size in GP	20−50
Generations in GP	100−2000



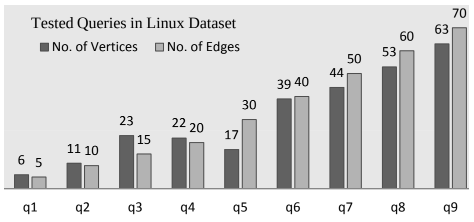


Fig. 9. Datasets, Queries and Parameter Settings

representation of the control and data flow dependency within a procedure, in which one vertex is assigned to each statement and an edge represents the dependency between two statements. PDG is widely used for software clone detection, optimization, debugging etc [12].

Query Graphs: We have selected a set of graphs from our dataset as base queries, which follow the distribution of the dataset. The basis of the selection is to create both similar and dissimilar set of answer graphs and to vary the initial distance bound in the expected answer set. For our case study, we have selected a set of graphs in AIDS dataset whose screening results are confirmed.

Answer and Expected (why-not) Graphs: The answer graphs are the graphs returned by the similar graph searching system. We have implemented SEGOS-like indexing method developed in [9] to manage our graph data and $TOP_k(q, D)$ querying system developed in [8] for matching similar graphs in D, which exploits the lower bound of distances, denoted by $\underline{\lambda}(q, g_i)$, between the query graph q and $g_i \in D$ given as follows.

$$\underline{\lambda}(q, g_i) = |E(q)| + |E(g_i)| - \max_{P} \sum_{s_q \in S(q)} \Lambda(s_q, P(s_q)) \quad (13)$$

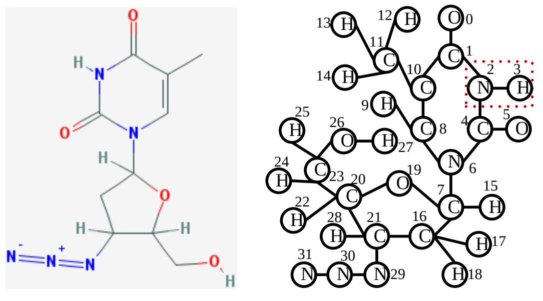
The why-not graphs are selected from $TOP_{2k}(q, D) \setminus TOP_k(q, D)$, i.e., the next top-$k$ similar graphs are selected as the missing targets in our experiments.

Our System: The proposed *why-not* question answering system is developed on top of the similar graph searching system. A summary about the datasets, queries and parameters used in experiments is given in Fig. 9.

7.2 Empirical Study

In this section, we empirically validate the usefulness of answering *why-not question* in similar graph matching via query graph modifications. More specifically, we analyze the suitability of our approach in relation to the modification of the initial query graph and thereafter, the additional resultant graphs that appear in the new answer set. We consider the followings: (1) a missing graph which is very similar to the answer set but currently missing from it; and (2) a far distant missing graph which is believed to be missing by a naive user.

Case Study-1: This case study demonstrates the usefullness of our approach by conducting an experiment in AIDS dataset [33]. Consider a molecular query graph "Zidovudine" (formula: $C_{10}H_{13}N_5O_4$), which is known to be "active" from its antiviral screening study. The 2D structure and the graph representation (32 vertices with 4 unique vertex labels) of this molecule is shown in Fig. 10. Suppose an inquisitive drug designer wishes to explore similar molecules g_i (might be active too)



(a) 2D structure (b) the graph

Fig. 10. Zidovudine

by issuing a similarity search, $\lambda(\text{"}Zidovudine\text{"}, g_i) \leq 2$, in the graph database D. After inspecting the results, the designer gets surprised by not receiving "Thymidine ($C_{11}H_{15}N_5O_4$ and active)", but "1-Adfat ($C_{10}H_{12}FN_5O_5$ and inactive)" in her answer set, i.e., She wonders "Why-not Thymidine"? She then asks to modify her initial hypothetical query "Zidovudine" to retrieve "Thymidine".

The $\overline{\lambda}_m$ of the expected answer set with the initial query graph is 5. That is, the user could end up retrieving "Thymidine" by issuing the query $\lambda(\text{"}Zidovudine\text{"}, g_i) \leq 5$ by *trial and error* process, which could be a never ending process. Now, we modify the hypothetical initial query to retrieve "Thymidine" by deleting the edge between vertices 2 and 3 of "Zidovudine". The $\overline{\lambda}_m$ of the modified query graph, $\lambda(\text{"}Zidovudine\text{"}', g_i) \leq 4$, with the expected answer set becomes 4 and the modified query retrieves "Thymidine" successfully in the new answer set. The modified query explains that the designer could retrieve "Thymidine" in her answer set if she would not have an edge between vertices 2 and 3 in her initial query. It should be noted that the modified query graph may not represent real-world molecules as there are restrictions on the formulation of bonds between different atoms, which can be studied further by considering the domain knowledge of the specific database. A more detail information about the above can be found in the supplemental material.

Case Study-2: Our approach modifies query graph via edge insertion/deletion to explain missing answer(s). Therefore, the modified query graph may end up bringing some graphs in the new answer set that were not present in the initial answer set. For example, the modified query $\lambda(\text{"}Zidovudine\text{"}', g_i) \leq 4$ in *Case Study-1* returns molecules "AZEU", "Azidothymidine" and "Uridine" in addition of "Thymidine" in the new answer set. Though, these molecules are very similar to "Thymidine", only "AZEU", and "Azidothymidine" are active. In [3], [34], [15], these additional resultant objects are termed as *false positives* (FP). Also, if any of the initial

TABLE 1
False Positive Rates (FPR) and Positive Predictive Values (PPV) in AIDS Dataset

Queries	**TOP$_5$+6**			**TOP$_5$+7**			**TOP$_5$+10**			**TOP$_{10}$+11**			**TOP$_{10}$+15**			**TOP$_{10}$+17**			**TOP$_{10}$+20**		
	$\overline{\lambda}(q,q')$	FPR	PPV	$\overline{\lambda}(q,q')$	FPR	PPV	$\overline{\lambda}(q,q')$	FPR	PPV	$\overline{\lambda}(q,q')$	FPR	PPV	$\overline{\lambda}(q,q')$	FPR	PPV	$\overline{\lambda}(q,q')$	FPR	PPV	$\overline{\lambda}(q,q')$	FPR	PPV
q_1	0	0.00016	0.50	0	0.00016	0.50	2	0.00023	0.41	2	0.00025	0.67	1	0.00012	0.71	1	0.00007	0.80	2	0.00012	0.71
q_2	3	0.00061	0.21	3	0.00061	0.21	3	0.00026	0.39	3	0.00016	0.63	3	0.00016	0.63	3	0.00016	0.63	2	0.00059	0.33
q_3	0	0.00002	0.88	3	0.00023	0.41	4	0.00009	0.63	1	0.00059	0.41	1	0.00059	0.41	2	0.00044	0.59	2	0.00044	0.59
q_4	4	0.00012	0.58	2	0.00007	0.70	4	0.00023	0.62	4	0.00023	0.62	4	0.00023	0.62	3	0.00009	0.85	3	0.00009	0.85

TABLE 2
Avg. Qualities in Linux Dataset, $|X| = 3 - 8$: Our Approach vs. Median Graph

Queries	**Avg.** $\overline{\lambda}_m$	**GS**	**GP$_{30,300}$**	**GP$_{30,500}$**	**BP**	**BT**	**MG**
q_1	2.0	0.750000	0.750000	0.750000	0.750000	0.750000	0.50000
q_2	12.0	0.583333	0.583333	0.583333	0.583333	0.583333	0.48611
q_3	17.6	0.458333	0.458333	0.458333	0.458333	0.458333	0.28611

TABLE 3
Avg. Qualities in AIDS Dataset, $|X| = 3 - 10$: Our Approach vs. Median Graph

Queries	**Avg.** $\overline{\lambda}_m$	**GS**	**GP$_{30,100}$**	**GP$_{30,500}$**	**BP**	**BT**	**MG**
q_1	6.3	0.31250	0.31250	0.31250	0.31250	0.31250	0.218750
q_2	14.8	0.28141	0.28974	0.28975	0.29808	0.29808	0.188717
q_3	14.4	0.30240	0.30240	0.30240	0.30240	0.30240	0.209090
q_4	22.0	0.34659	0.37500	0.37500	0.37500	0.37500	0.346598

resultant objects are excluded from the new answer set, then it is termed as *false negative* (FN) [34], [15].

To demonstrate the effect of including missing answer(s) in the new answer set of the modified query graph, we conduct experiments in the AIDS dataset again with varying degree of distances between the initial query graphs $q_1 - q_4$ (Fig. 9) and the missing answer(s). Then, we calculate the distance between the original query graph q and the new query graph q' ($\overline{\lambda}(q, q')$, the lower the better), the *false positive rate* ($FPR = \frac{FP}{FP+TN}$, the lower the better) and *positive predictive value* or *precision* ($PPV = \frac{TP}{TP+FP}$, the greater the better) for each tested query, where TN and TP are the *true negatives* and *true positives*, respectively. The results are shown in Table 1. The second column in Table 1 represents that we consider TOP-5 similar graphs as the initial answer set and the sixth most similar database graph is taken as the *missing* or *why-not* graph. This holds similarly for other columns (column 3 to column 8). The value 0 (zero) for $\overline{\lambda}(q, q')$ signifies that our approach does not achieve any refinement for the user submitted query graph. From Table 1, we observe that our approach is correct up to three decimal places in terms of *false positive rates* for the tested queries $q_1 - q_4$. In general, we achieve better $precision$ (PPV) for lower $\overline{\lambda}(q, q')$ w.r.t. a certain query setting. However, we do not observe any definite relationship between *false positive rates* (or *precision*) and the distance of the *why-not* graph to the initial query graph in our approach. We believe that these metrics are dependent on the similarities of the *why-not* graph and the database graphs with the initial query graph, which require further study together with the domain knowledge of the specific database.

7.3 Effectiveness Study

This section presents the effectiveness study of different candidate selection algorithms: BP, GS and GP proposed in this paper in terms of the quality improvements (Eq. 10) in the modified queries. To do so, we create $(N_2 - N_1) + 1$ instances of each query (Fig. 9) with varying size of expected answer set X from $N_1, N_1+1, ..., N_2$ and report the average of the quality improvements in them.

Our Approach (BP, GS, GP) vs. Median Graph: We compare our approach with the generalized median graph (MG) computed by applying exhaustive search [27], which is exact. The MG does not work for answering *why-not* questions in similar graph matching as we see from Table 2 and Table 3. The qualities offered by MG are not comparable to our approach. This is because MG is designed to minimize the SOD, not the distance bound λ_m. Two solutions may have the same SOD with different distance bounds. We minimize the overall distance bound for the new query graph and exploits SOD to break the tie only.

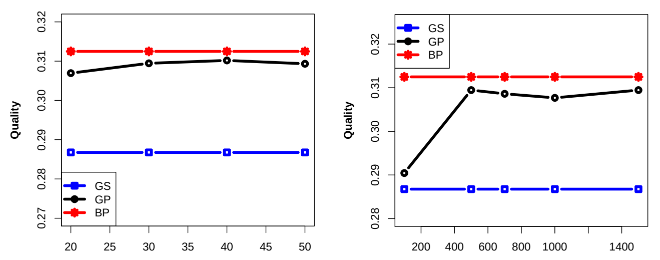
TABLE 4
Avg. Qualities for Different Candidate Selection Algorithms in AIDS Dataset, $|X| = 3 - 50$

Queries	**Avg.** $\overline{\lambda}_m$	**GS**	**GP$_{30,100}$**	**GP$_{30,500}$**	**GP$_{30,1000}$**	**GP$_{50,500}$**	**BP**
q_1	13.2	0.188454	0.197001	0.195868	0.198135	0.200403	0.202670
q_2	19.3	0.242145	0.267754	0.268596	0.269830	0.269830	0.269830
q_3	18.6	0.254558	0.259892	0.259892	0.259892	0.259892	0.259892
q_4	24.6	0.286755	0.290444	0.309453	0.307675	0.309332	0.312483

BP vs. BT based Candidate Selection: Our BP based candidate selection algorithm performs the same as backtracking without pruning (BT) algorithm does as shown in Table 2 and Table 3. This indicates that our pruning idea works for filtering some non-promising candidate nodes and their descendants (i.e., subtree in the search tree) without sacrificing the quality of the results in the new query graph.

GS vs. BP based Candidate Selection: The GS based candidate selection algorithm performs worse compared to BP in terms of the quality of the refined query graphs as we can observe it in Table 4. However, GS is executable for any number of graphs in X, i.e., GS is time efficient (which is explained in section 7.4). Therefore, GS is preferable to BP for large size of X and also if X has very dissimilar graphs measured by $\overline{\lambda}_m$ (which generates large number of candidates).

GP vs. BP based Candidate Selection: The performance of the proposed GP based candidate selection algorithm is comparable to the performance offered by BP as we observe it in Table 4. However, the performance of GP depends on its specific parameter settings (e.g., population size, number of generations etc) as shown in Table 4. For example, the average performance of the query graph q_4 (in AIDS dataset) reaches to the



(a) Vary population size (b) Vary number of generations

Fig. 11. Effect of GP parameters on the quality of new q_4 in AIDS Dataset

TABLE 5
Avg. Qualities in Linux Dataset,$|X|$=3-50: GS vs. GP

Queries	**Avg. $\overline{\lambda}_m$**	**GS**	**GP$_{x,500}$**			**GP$_{30,y}$**		
			x=30	x=40	x=50	y=0.3K	y=1K	y=1.5K
q_4	23	0.449222	0.459264	0.459264	0.459264	0.459264	0.459264	0.459264
q_5	9	0.496806	0.496806	0.496806	0.496806	0.496806	0.496806	0.496806
q_6	50	0.693618	0.703422	0.7042391	0.703831	0.702197	0.703422	0.702605
q_7	63	0.298091	0.304409	0.304694	0.305691	0.276758	0.308012	0.306643
q_8	74	0.796367	0.796367	0.796367	0.796367	0.796367	0.796367	0.796367
q_9	96	0.385417	0.381511	0.383898	0.382813	0.379557	0.381728	0.382379

performance offered by BP if we increase the population size for fixed number of generations or if we increase the number of generations for fixed sized population in GP based candidate selection (as shown in Fig. 11).

GS vs. GP based Candidate Selection: The GP based candidate selection algorithm performs better compared to the GS based candidate selection scheme in general as shown in Table 4. However, GP may not always outperform GS if GP does not start with a good set of chromosomes in its initial population as we see it for q_7 and q_9 in Table 5 and q_7, q_8, q_9 and q_{10} in Table 6. To overcome this problem in GP, we can initialize GP with the outcome of GS (i.e., a number of chromosomes in the initial population of GP are initialized with the GS outcome) and guarantee that GP will perform at least as GS does (as shown in Table 7 and Table 8). This initialization technique can also boost the performance of GP further. For example, the maximum quality achieved in the refined query of q_6 of Linux dataset with GP is 0.7042391 (see Table 5), which has been improved to 0.706282 (see Table 7) if we initialize 25% chromosomes of the initial population of GP with the GS outcome.

Summary: GP performs in between GS and BP w.r.t. a specific parameter settings. However, if there is less dissimilar graphs in the set X and X does not include too many graphs, then we can consider to run BP based candidate selection scheme for the best solution. Otherwise, we can consider to run GP for trading off the qualities of the modified query graphs by initializing it with the GS outcome.

TABLE 6
Avg. Qualities in AIDS Dataset, $|X|$=3-50: GS vs. GP

Queries	**Avg. $\overline{\lambda}_m$**	**GS**	**GP$_{x,0.5K}$**			**GP$_{30,y}$**		
			x=30	x=40	x=50	y=1K	y=1.5K	y=2K
q_5	29.9	0.328014	0.333412	0.331371	0.333412	0.331371	0.333412	0.333412
q_6	52.8	0.382244	0.389226	0.390333	0.392186	0.388009	0.391022	0.389347
q_7	76.4	0.438244	0.432605	0.431096	0.430699	0.450055	0.450587	0.450855
q_8	96.2	0.455119	0.409370	0.409153	0.408101	0.431339	0.469079	0.454882
q_9	117	0.412518	0.256405	0.258495	0.25519	0.424146	0.439277	0.441056
q_{10}	138	0.465823	0.190820	0.192868	0.183325	0.431339	0.469079	0.471869

TABLE 7
Avg. Qualities in Linux Dataset, $|X| = 3 - 50$: GS vs. GP (25% population in GP is initialized with GS outcome)

Queries	**Avg. $\overline{\lambda}_m$**	**GS**	**GP$_{x,0.5K}$**			**GP$_{30,y}$**		
			x=30	x=40	x=50	y=1K	y=1.5K	y=2K
q_6	50	0.693618	0.702605	0.702197	0.702605	0.705465	0.705056	0.706282
q_7	63	0.298091	0.307961	0.308871	0.310194	0.308555	0.307253	0.309217
q_8	74	0.796367	0.796367	0.796367	0.796367	0.796367	0.796367	0.796367
q_9	96	0.385417	0.385417	0.385417	0.385417	0.385417	0.385417	0.385417

TABLE 8
Avg. Qualities in AIDS Dataset, $|X| = 3 - 50$: GS vs. GP (25% population in GP is initialized with GS outcome)

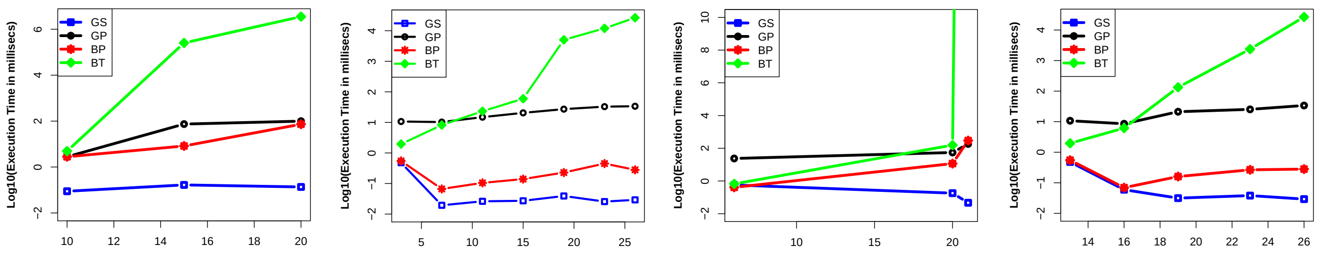
Queries	**Avg. $\overline{\lambda}_m$**	**GS**	**GP$_{x,0.5K}$**			**GP$_{30,y}$**		
			x=30	x=40	x=50	y=1K	y=1.5K	y=2K
q_7	76.4	0.438244	0.448081	0.449659	0.449880	0.451922	0.451655	0.452189
q_8	96.2	0.455119	0.455330	0.455330	0.455330	0.455967	0.455750	0.455961
q_9	117.1	0.412518	0.437939	0.455330	0.438111	0.439531	0.440942	0.441466
q_{10}	138	0.465823	0.469073	0.469073	0.469373	0.469520	0.471575	0.471282

7.4 Efficiency Evaluation

This section evaluates the performances of the proposed BP, GS and GP based candidate selection algorithms for answering *why-not* questions in similar graph matching. More specifically, we have compared the performances of these algorithms by analyzing the execution times of different input queries with varied size, varied size of X, varied $\overline{\lambda}_m$ and varied candidate set size.

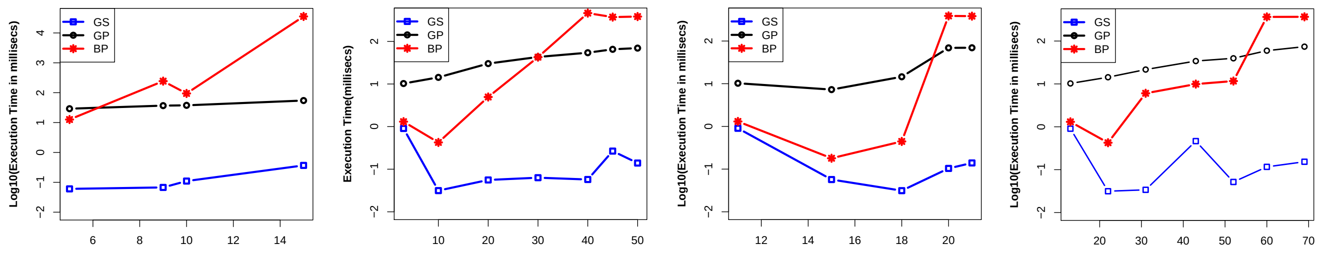
BT vs. GS, GP and BP based Candidate Selection: The BT based candidate selection algorithm is time-inefficient compared to the GS, GP and BP based candidate selection algorithm. Because, BT performs an exhaustive search in the space $2^{|O|}$. If the query size as well as the number of graphs in X increases (which may result in increased number of candidates in O), the execution time of BT increases a lot (exponentially) compared to BP and the other candidate selection algorithms as shown in Fig. 12(a)- Fig. 12(c). As an example, for query graph q_2 and $|X|$=26 in Linux dataset, we observe BT takes around 27 seconds to finish where BP takes only 2.8 milliseconds. Similarly, BT also takes much longer time compared to the other candidate selection algorithms proposed in this paper if the number of candidates increases in O as shown in Fig. 12(d). Therefore, BT is not time efficient for searching the suitable set of candidates and thereby refine the initial query graph.

BP vs. GS and GP based Candidate Selection: The BP based candidate selection algorithm takes much longer time compared to GS and GP based candidate selection algorithms proposed in this paper if (a) query size increases, (b) the number of graphs increases in X, (c) X includes very dissimilar graphs, and (d) the number candidate operations in O increases (as shown in Fig. 13). These reasons are not independent from each other, rather they may complement each other. There is no monotonic relationship between them. For certain query settings, BP may take hours to finish if it can not prune sufficient number of subtrees (performs the same as BT). However, BP may finish its execution quickly if the set X includes very similar graphs (small $\overline{\lambda}_m$) and the query size (i.e., $|E(q)|$) is small. This can be observed from



(a) Effect of query size, $q_2 - q_4$ (b) Effect of size of X for q_2 (c) Effect of $\overline{\lambda}_m$ for q_3 (d) Effect of candidate size for q_2

Fig. 12. Efficiency in Linux Dataset: BT vs. GS, GP and BP based candidate selections



(a) Effect of query size, $q_1 - q_4$ (b) Effect of size of X for q_3 (c) Effect of $\overline{\lambda}_m$ for q_3 (d) Effect of candidate size for q_3

Fig. 13. Efficiency in AIDS Dataset: BP vs. GS and GP based candidate selections

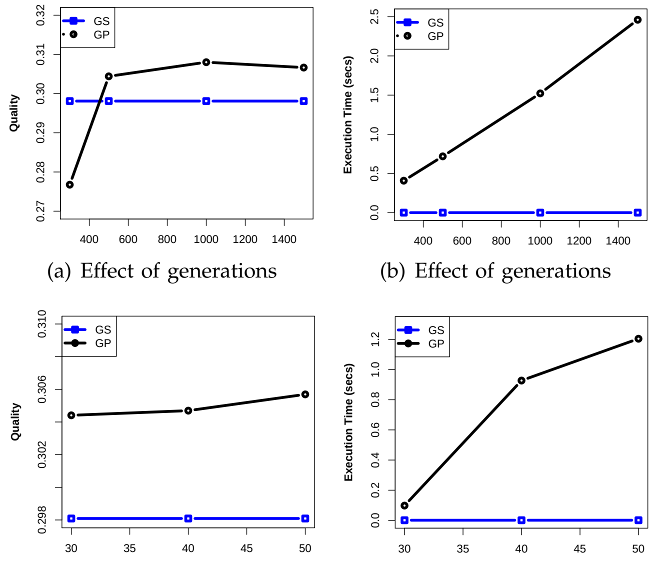
Fig. 12 where BP takes less time compared to the GP based candidate selection (in Fig. 12 (b) and Fig. 12 (d) we have experimented query q_2 for Linux dataset where $|E(q_2)| = 10$ and average $\overline{\lambda}_m = 14.5$).

GS vs. GP and BP based Candidate Selection: The GS based candidate selection algorithm takes less time compared to GP and BP based candidate selection algorithms proposed in this paper as shown in Fig. 13. Therefore, GS is very time efficient. However, we may need to sacrifice the qualities in the refined queries for this time-efficiency as explained in Section 7.3.

GP vs. GS and BP based Candidate Selection: The efficiency of GP based candidate selection algorithm depends on its specific parameter settings, e.g., population size, number of generations and their combinations. In general, GP takes much longer time than GS based candidate selection algorithm as we see in Fig. 13. Also, if X includes very dissimilar set of graphs and $|X|$ is large (which results in increased number of candidates in O and increased execution time for BP), then GP takes less time compared to the BP as shown Fig. 13.

Effect of Parameter Settings in GP: In GP, we can trade-off the quality of the results as we want by setting its parameter accordingly (as explained in Section 7.3). That is, we need to increase the population size and/or the number of generations in GP based candidate selection to increase the qualities of the refined queries (i.e., to achieve qualities comparable to BP). However, if we increase the population size and/or the number of generations, the execution time in GP also increases accordingly (as shown for q_7 of Linux dataset in Fig. 14).

Summary: The efficiency of GP lies in between GS and BP based candidate selection algorithms for large size of X and O. Therefore, we advocate to use BP for small size of graphs in X (to achieve better quality in q^*) and to use GP for large size of graphs (for which BP is inefficient)



(a) Effect of generations (b) Effect of generations

(c) Effect of population size (d) Effect of population size

Fig. 14. Effect of parameter settings in GP for query q_7 in Linux Dataset

in X by initializing it with the GS outcome.

8 Conclusion and Future Work

This paper addresses the problem of answering why-not questions in similar graph matching for graph databases. To address the problem, we have proposed an approximate solution approach as computing the exact solution is NP-hard. We have also established the search space for the new query graph. Our solution starts with finding the approximate maximum common subgraph between the initial query and the expected answer set as well as the distances to be minimized for the new query graph. To minimize the upper distance bound between the new query and the answer set, we have proposed candidate edge generation and three different candidate selection algorithms for modifying the initial query graph. Finally, we have conducted extensive experiments, and confirmed the effectiveness and efficiency of our approach.

The proposed framework can be explored further as follows: (1) it can be extended for generalized graph distance function by enhancing vertex mapping and customizing edge selection algorithms accordingly; and (2) the quality function can be enhanced to incorporate measures such as false positives if the labels of the graphs are known (to make it more domain specific).

Acknowledgment

We would like to sincerely thank the reviewers for their useful suggestions and Dr. Benjamin A. Dalton for helping us conduct the case studies. The work is supported by the Australian Research Council (ARC) discover project DP140103499.

References

[1] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu, "Making database systems usable," in *SIGMOD Conference*, 2007, pp. 13–24.

[2] A. Chapman and H. V. Jagadish, "Why not?" in *SIGMOD Conference*, 2009, pp. 523–534.

[3] Q. T. Tran and C.-Y. Chan, "How to conquer why-not questions," in *SIGMOD Conference*, 2010, pp. 15–26.

[4] Z. He and E. Lo, "Answering why-not questions on top-k queries," in *ICDE*, 2012, pp. 750–761.

[5] M. S. Islam, R. Zhou, and C. Liu, "On answering why-not questions in reverse skyline queries," in *ICDE*, 2013, pp. 973–984.

[6] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Comput. Surv.*, vol. 40, no. 1, pp. 1:1–1:39, Feb. 2008.

[7] C. C. Aggarwal and H. Wang, Eds., *Managing and Mining Graph Data*, ser. Advances in Database Systems. Springer, 2010, vol. 40.

[8] Y. Zhu, L. Qin, J. X. Yu, and H. Cheng, "Finding top-k similar graphs in graph databases," in *EDBT*, 2012, pp. 456–467.

[9] X. Wang, X. Ding, A. K. H. Tung, S. Ying, and H. Jin, "An efficient graph indexing method," in *ICDE*, 2012, pp. 210–221.

[10] Y. C. Martin, J. L. Kofron, and L. M. Traphagen, "Do structurally similar molecules have similar biological activity?" *J. Med. Chem.*, vol. 45, pp. 4350–4358, 2002.

[11] E. Karakoç, A. Cherkasov, and S. C. Sahinalp, "Distance based algorithms for small biomolecule classification and structural similarity search," in *ISMB (Supplement of Bioinformatics)*, 2006, pp. 243–251.

[12] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE*, 2008, pp. 321–330.

[13] Y. Zhu, L. Qin, J. X. Yu, Y. Ke, and X. Lin, "High efficiency and quality: large graphs matching," *VLDB J.*, vol. 22, no. 3, pp. 345–368, 2013.

[14] J. Huang, T. Chen, A. Doan, and J. F. Naughton, "On the provenance of non-answers to queries over extracted data," *PVLDB*, vol. 1, no. 1, pp. 736–747, 2008.

[15] M. S. Islam, C. Liu, and R. Zhou, "Flexiq: A flexible interactive querying framework by exploiting the skyline operator," *Journal of Systems and Software*, vol. 97, pp. 97–117, 2014.

[16] E. B. Krissinel and K. Henrick, "Common subgraph isomorphism detection by backtracking search," *Softw. Pract. Exper.*, vol. 34, no. 6, pp. 591–607, May 2004.

[17] F. N. Abu-Khzam, N. F. Samatova, M. A. Rizk, and M. A. Langston, "The maximum common subgraph problem: Faster solutions via vertex cover," in *AICCSA*, 2007, pp. 367–373.

[18] I. Koch, "Enumerating all connected maximal common subgraphs in two graphs," *Theor. Comp. Sci.*, vol. 250, no. 1-2, pp. 1–30, 2001.

[19] J. W. Raymond, E. J. Gardiner, and P. Willett, "Rascal: Calculation of graph similarity using maximum common edge subgraphs," *Comput. J.*, vol. 45, no. 6, pp. 631–644, 2002.

[20] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. V. Dooren, "A measure of similarity between graph vertices: Applications to synonym extraction and web searching," *SIAM Rev.*, vol. 46, no. 4, pp. 647–666, Apr. 2004.

[21] T. Caelli and S. Kosinov, "An eigenspace projection clustering method for inexact graph matching," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 4, pp. 515–519, Apr. 2004.

[22] M. Zaslavskiy, F. R. Bach, and J.-P. Vert, "Global alignment of protein-protein interaction networks by graph matching methods," *Bioinformatics*, vol. 25, no. 12, 2009.

[23] X. Yan, P. S. Yu, and J. Han, "Substructure similarity search in graph databases," in *SIGMOD Conference*, 2005, pp. 766–777.

[24] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan, "Nema: Fast graph search with label similarity," *PVLDB*, vol. 6, no. 3, pp. 181–192, 2013.

[25] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas, "Exemplar queries: Give me an example of what you need," *PVLDB*, vol. 7, no. 5, pp. 365–376, 2014.

[26] X. Jiang, A. Münger, and H. Bunke, "On median graphs: Properties, algorithms, and applications," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 23, no. 10, pp. 1144–1151, 2001.

[27] M. Ferrer, E. Valveny, and F. Serratosa, "Median graph: A new exact algorithm using a distance based on the maximum common subgraph," *Pattern Recognition Letters*, vol. 30, no. 5, pp. 579–588, 2009.

[28] R. Raveaux, S. Adam, P. Héroux, and É. Trupin, "Learning graph prototypes for shape recognition," *Computer Vision and Image Understanding*, vol. 115, no. 7, pp. 905–918, 2011.

[29] M. Ferrer, D. Karatzas, E. Valveny, I. Bardají, and H. Bunke, "A generic framework for median graph computation based on a recursive embedding approach," *Computer Vision and Image Understanding*, vol. 115, no. 7, pp. 919–928, 2011.

[30] M. Ferrer, E. Valveny, F. Serratosa, I. Bardají, and H. Bunke, "Graph-based *k*-means clustering: A comparison of the set median versus the generalized median graph," in *CAIP*, 2009, pp. 342–350.

[31] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics*, vol. 2, p. 8397, 1955.

[32] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998.

[33] Aids antiviral screen. [Online]. Available: http://dtp.nci.nih.gov/docs/aids/aids_data.html

[34] M. S. Islam, C. Liu, and R. Zhou, "A framework for query refinement with user feedback," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1580–1595, 2013.

Md. Saiful Islam received his BSc (Hons) and MS degree in Computer Science and Engineering from University of Dhaka, Bangladesh, in 2005 and 2007, respectively. He finished his PhD in 2014 at the Swinburne University of Technology, Australia. He is currently a postdoctoral research fellow at Swinburne University of Technology. His current research interests are in the areas of data management, information retrieval, machine learning and computer architecture. He is a member of IEEE and ACM.

Chengfei Liu received the BS, MS and PhD degrees in Computer Science from Nanjing University, China in 1983, 1985 and 1988, respectively. Currently, he is a Professor in Swinburne University of Technology, Australia. His research interests include keywords search on structured data, query processing and refinement for advanced database applications, query processing on uncertain data and big data, and data-centric workflows. He is a member of IEEE and ACM.

Jianxin Li received his BE and ME degrees in computer science, from the Northeastern University, China, in 2002 and 2005, respectively. He received his PhD degree in computer science, from the Swinburne University of Technology, Australia, in 2009. He was a research fellow in Swinburne from 2009 to 2014. He is currently a senior research fellow at RMIT University, Australia. His research interests include database query processing and optimization, and social network analytics.