Knowledge-Based Systems 72 (2014) 48–59

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Graph indexing for large networks: A neighborhood tree-based approach

Zhen Lin [a,b,*], Yijun Bei [c]

[a] College of Computer Science & Technology, Zhejiang University, Hangzhou 310027, China
[b] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
[c] School of Software Technology, Zhejiang University, Hangzhou 310027, China

ARTICLE INFO

Article history:
Received 17 March 2014
Received in revised form 4 August 2014
Accepted 29 August 2014
Available online 16 September 2014

Keywords:
Graph querying
Graph matching
Neighborhood tree
Indexing
Social network

ABSTRACT

Graphs are used to model complex data objects and their relationships in the real world. Finding occurrences of graph patterns in large graphs is one of the fundamental graph analysis tools used to discover underlying characteristics from these complex networks. In this paper, we propose a new tree-based approach for improving subgraph-matching performance. First, we introduce a new graph indexing mechanism known as **N**eighborhood **Tree**s (NTree), which records the neighborhood relationships of each vertex in the large graph to filter negative vertices. Second, we decompose a query graph into a set of neighborhood trees and only a subset of candidate trees, which can properly recover the original query graph. In this way, the tree-at-a-time method is used to obtain the matched graphs. Third, we employ a graph query optimizer to determine the neighborhood tree selection order on the basis of the cost evaluation of tree join operations. Experiments on both real and synthetic databases demonstrate that our approach is more efficient than other state-of-the-art indexing methods.

1. Introduction

Graphs are used to model many complex data objects and their relationships in the real world. In recent years, an increasing number of large networks such as social networks, chemical compounds, semantic web, and protein networks have appeared. The sizes of these large graphs can be in excess of millions of vertices and edges. Methods for managing, processing, and analyzing these graph data have become important research topics. Finding the occurrences of graph patterns or subgraphs in large graphs is one of the fundamental uses for graph analysis tools because they denote underlying characteristics in complex networks. Subgraph query or matching approaches have been widely used in areas such as chemical informatics [1], proteins analysis [2], biochemistry [3], web applications [4], and computer vision [5]. For example, given a large protein network, biologists may want to determine all occurrences of structural motifs in 3D proteins by using protein contact maps [6].

Graph searching is an important task in a variety of applications and falls under two scenarios: subgraph querying and subgraph matching. The classical graph-querying problem is to find all supergraphs of the query pattern from a graph database, whereas graph matching involves finding all subgraphs of the database graph, which are isomorphic to the query graph. It is clearly inefficient to perform an exhaustive search in the database because the subgraph isomorphism itself is a non-deterministic polynomial time (NP)-complete problem.

Graph indexing is a common technique for performing searches in large graph databases. Many indexing mechanisms such as gIndex [7], Tree+Δ [8], TreePi [9], FERRARI [10], and frequent subgraph (FG)-index [11] have been developed for subgraph querying processes. In these approaches, frequent features are extracted from graphs and are leveraged to build graph indices. As a result, expensive preprocessing is required due to the frequent pattern mining processes in index construction such as paths, trees, and subgraphs. In addition, non-mining based techniques, such as Closure-tree [12], GCoding [13], LnGCoding [14], and GiS [15], have been proposed for graph queries. For the purpose of indexing, structural information of graphs are mapped into graph signatures such as numerical space in GCoding or line graphs in GiS. However, such indexing was developed mainly for searching graphs in a large number of small graphs.

To solve the problem of graph matching in a large graph, neighborhood signature-based techniques such as GraphQL [16], SPath [17], and GADDI [18] have been proposed. In GraphQL [16], profiles

* Corresponding author at: College of Computer Science & Technology, Zhejiang University, Hangzhou 310027, China.
E-mail addresses: linz@illinois.edu, nblinz@gmail.com (Z. Lin), byj@zju.edu.cn (Y. Bei).

around a vertex neighborhood are used for local pruning, and global structural information is leveraged to simultaneously reduce the overall search space. In SPath [17], the shortest paths around a vertex neighborhood are leveraged as basic indexing units, and a more efficient path-at-a-time method is introduced to process graph queries. In GADDI [18], neighboring discriminating substructure (NDS) distance measurement is adopted as the basis of the pruning method, and an index-based graph matching method is proposed for achieving high pruning power and linear size scales. To reduce search space, the cost model and query plan optimizer are both employed in these up-to-date approaches. However, when using neighborhood profiles or paths as indexing units to prune negative vertices, structural information around the vertices may be lost. As a result, false positive vertices remain as candidates and require further filtering. Moreover, too many join operations are required compared with more complex structures such as trees or subgraphs.

In this paper, we propose a new tree-based approach for improving subgraph searching performance. First, we introduce a new graph-indexing mechanism known as **N**eighborhood **Tree**s (NTree), which records the neighborhood relationships of each vertex in the large graph to filter negative vertices. Because trees contain more information than paths or vertex profiles, the NTree has stronger pruning power. Second, we decompose a query graph into a set of trees, and only a subset of candidate subtrees that can properly recover the original query graph are selected. These candidate subtrees are then joined to reconstruct the query graph. In this way, the tree-at-a-time method is used to obtain the matched graphs. Third, we employ a graph query optimizer to determine the vertex searching order on the basis of the cost evaluation of tree joining operations. Our work has the following contributions:

1. We propose a structural pattern-based graph-matching framework. First, candidate vertices are identified. Then, a joining process by comparing relationships of candidate vertices in the large graph to the original query graph is used to further verify these candidates. Both vertex pruning and query reconstructing abilities of different structural patterns such as neighborhood paths, trees, and graphs are evaluated. As a result, neighborhood trees are selected to be the most feasible candidates (Section 3).
2. To reduce the number of candidates, we propose a new cost-effective graph indexing technique, NTree, which makes use of trees around the vertex neighborhood for pruning purposes. Canonical unordered trees are leveraged, and the string comparison technique is used to accelerate the subtree containment process (Section 4).
3. To address the second step, we propose an efficient searching method for joining the neighborhood trees and reconstructing the original query graph. In addition, we design a graph query cost model on the problem of neighborhood tree selection to optimize the search order (Section 5).
4. We conduct extensive experiments by using both real and synthetic databases. We compare our indexing method, NTree, with state-of-the-art path-based indexing methods. The results show that our method outperforms these indexing methods in terms of graph matching performance (Section 6).

2. Related works

The subgraph isomorphism test [19,20,5,21] is a well-known NP-complete problem that has been widely studied in recent years. For subgraph searching, a large number of index-based graph matching and searching frameworks have been proposed including gIndex [7], TreePi [9], FG-index [11], NB-index [22], LW-index [23], Tree+Δ [8], GCode [13], GPTree [24], Closure-tree [12], Turboiso [25], SODA [26], SING [27], and GiS [15]. These graph-indexing approaches have been designed mainly for performing subgraph querying from a graph database consisting of many small- or medium-sized graphs. Most of these indexing approaches such as gIndex [7], Tree+Δ [8], and FG-index [11] all make use of frequent patterns in the graph database as the basic indexing structure. Consequently, expensive preprocessing is required to mine frequent patterns when constructing graph indices. The encoding method GCode [13] assigns a signature known as level-n path tree to each vertex based on its local structures. As a result, graph codes are obtained by combining all vertex signatures. SING [27] uses the concept of features and makes use of feature locality information. gStore [28] transforms a Resource Description Framework (RDF) graph into a data signature graph and uses the vertex signature (VS)*-tree index with light maintenance overhead. A filtering rule in gStore is also developed for answering exact SPARQL queries in a uniform manner. In this paper, a different encoding scheme is introduced. We make use of the level-wise signature that records the neighborhood edge to construct the neighborhood tree as the graph index. Neighborhood vertices arranged in the form of rings are recorded as well as the occurring number of edges in each level.

To address the inexact matching problem, Grafil [29] clusters features according to their selectivities and applies a multi-filter strategy. Edge deletions are transformed into feature misses, and an upper bound is used on the maximum number of allowed feature misses for graph filtering. Zhu et al. [30] proposed a novel search paradigm, TreeSpan, to conduct similarity all-matching that conforms a similarity threshold θ by first conducting exact all-matching on a minimal set of spanning trees. A rigid theoretic analysis shows that this approach can significantly reduce the time required for conducting exact all-matching compared with the existing techniques. In SAGA [31], Tian et al. proposed a more flexible indexing approach that can support both vertex insertions and deletions. A flexible graph distance model is employed to measure similarities between graphs, and matched fragments are assembled into large matches. However, in this similar graph searching approach, only a subset of approximate matching results is obtained. Ness [32] is another tool for inexact matching that focuses on the $top-k$ approximate matches. In this method, a neighborhood-based similarity measure is proposed that avoids costly graph isomorphisms and edits distance computation. SLQ [33] is a framework enabling schemaless and structureless graph query that can automatically learn an effective ranking model with no manual preprocessing. This method returns matches by using graph sketches and belief propagation. The simB method [34] is proposed for edit distance-based similarity search problems, whereby a lower bound based on the branch structure is proposed to reduce the search space, and the b-tree index is adopted to facilitate the query processing. GSimSearch [35] is another efficient algorithm for graph similarity query. Unlike the simB method, GSimSearch exploits the number of common fixed-length paths between pairs of graphs and also adopts degree-associated structural information to enhance runtime performance.

In addition to the searching subgraph problem from a large number of small graphs, methods used to search a subgraph in a large graph such as a social network are also addressed [16,17,36,37,18,38,39]. Several up-to-date approaches such as GraphQL [16], SPath [17], and GADDI [18] are proposed to obtain all occurrences of a query in a large graph. In GraphQL [16], neighborhood profiles are first employed to prune vertices individually. Then, the overall search space by considering all vertices in the pattern is simultaneously reduced. In SPath [17], neighborhood signatures of vertices are used for vertex filtering. The decomposed shortest paths are considered as patterns to be joined, and the path-at-a-time method for graph matching is proposed. In GADDI [18], the NDS distance between pairs of neighboring vertices that satisfies the inequality property is indexed. Furthermore, the GADDI method also applies two-way pruning and incorporates a dynamic matching scheme. Sun et al. [37] used efficient in-memory graph exploration and massive parallel computing for subgraph matching rather than super-linear indices. Huang et al. [39] introduced two optimization frameworks based on dynamic programming and cycle detection for distributed graph pattern matching and also proposed a computation reuse technique to eliminate redundant subgraph pattern matching.

3. Preliminaries

In this section, we first provide basic definitions of graphs and outline the structural pattern-based algorithmic framework to address the graph matching problem. We then present a cost evaluation model upon which our analysis of different neighborhood patterns for graph indexing is based.

3.1. Problem statement

Definition 1 (*Graph*). A graph $G = (V, E, \Sigma, l)$ is defined as an undirected labeled graph where V is a set of vertices, E is a set of edges between vertices, Σ is a set of vertex labels, and l is a labeling function that defines the mapping $l : V \to \Sigma$.

Definition 2 (*Subgraph Isomorphism*). A graph G is subgraph isomorphic to a graph G', iff there exists an injective function $f : V(G) \to V(G')$, such that (1) $\forall u \in V(G), l(u) = l'(f(u))$ and (2) $\forall (u, v) \in E(G), (f(u), f(v)) \in E(G')$, where l and l' are the label functions of G and G'.

If there exists a subgraph isomorphism from G to G', then G is called a subgraph of G', denoted as $G \subseteq G'$. The injective function f is considered as a match of G in G', and an occurrence of G is obtained in G'.

Graph matching problem: Given large graph G and query graph Q, we aim to discover an injective mapping set $\mathcal{F}$ such that each element $f \in \mathcal{F}$ is an injective function, which represents that query graph Q is a subgraph isomorphic to graph G:

$$\mathcal{F} : Q \to Q' \subseteq G := \{f | f : V(Q) \Rightarrow V(Q')(\textit{Query graph } Q \textit{ is subgraph isomorphic to graph } G), \forall u \in V(Q) \wedge \exists v \in V(Q')\}, \quad (1)$$

where graph Q' is a subgraph of graph G and satisfies bijective mapping from query graph Q to graph Q', which is a subgraph isomorphic to graphs Q and G.

The graph matching problem is to find all occurrences of query graph Q in the large graph G. The formal representation is

$$\mathcal{H} := \{g | \forall g = Q' \subseteq G, \textit{satisfy injective functions } \mathcal{F} : Q \to Q'\}, \quad (2)$$

where $\mathcal{H}$ contains all of the subgraphs from graph G, which are subgraphs isomorphic to query graph Q.

Example 3.1. Fig. 1 illustrates a large graph G and a query graph Q. Numeric identifiers are used to distinguish vertices and edges, whereas capital letters are used to represent various types of vertex labels. The subgraph S of G colored in gray with vertices $V(S) = \{9, 8, 7, 6, 5, 14, 10\}$ is a matching of Q in G. Consequently, Q is a subgraph isomorphic to G, and we have $Q \subseteq G$. In some cases, there may exist several occurrences of Q in G. For example, if a query Q' is a triangle graph with vertices $V(Q') = \{1, 2, 3\}$, then graphs with vertices $\{5, 2, 4\}, \{5, 4, 2\}, \{9, 7, 8\}, \{9, 8, 7\}, \{11, 12, 13\}$, and $\{11, 13, 12\}$ are all matching results.

3.2. Pattern-based algorithmic framework

Definition 3 (*Neighborhood Pattern*). Given graph G, vertex u in G, and radius r, the neighborhood pattern p of node u that presents the neighborhood signature by vertices within r hops away from u is denoted as $p^r_G(u)$. A neighborhood path of G is a path with r hops from the node u. A neighborhood tree of G consists of all of the longest paths starting from u with distances no more than r. A neighborhood subgraph of G is constructed by all vertices within r hops away from u.

Example 3.2. Fig. 2 shows three different neighborhood patterns of vertex u_1 in query Q. The neighborhood subgraph contains more structural information compared with the other structural patterns. It should be noted that for a neighborhood path and tree of the same vertex, different patterns may exist.

Definition 4 (*Effective Neighborhood Pattern*). Given large graph G, query graph Q, and radius r, Q is a subgraph isomorphic to G with mapping f if there exists a neighborhood pattern p. Then, we have $\forall u \in V(Q), p^r_Q(u) \subseteq p^r_G(f(u))$, where $f(u) \in G$. This neighborhood pattern is considered effective and can be leveraged to prune negative vertices.

Example 3.3. If a neighborhood subgraph of G consists of all vertices within distance r and all edges between the vertices, then it appears to be an ideal neighborhood pattern. Fig. 2(c) shows a neighborhood-induced subgraph *NGraph* of vertex u_1, which preserves local structural information. However, because the neighborhood subgraph has the ability of recording maximum graph information, it is rarely used to compute the graph containment relationship. As a result, the neighborhood subgraph is not suitable for indexing.

Definition 5 (*Candidate Vertices*). Given graph G, query Q, vertex $u \in V(Q)$, radius r, and neighborhood pattern p, the candidate vertices of u is the set of vertices in G that satisfies the neighborhood pattern ϕ_u:

$$C(u) = \{v | v \in V(G), \phi_u(v) = true\}, \quad (3)$$

where $\phi_u(v)$ means $p^r_Q(u) \subseteq p^r_G(v)$. All matched neighborhood patterns in G for u are defined as

$$C(p^r_Q(u)) = \{p^r_G(v) | p^r_G(v) \subseteq G, v \in C(u)\}. \quad (4)$$

Example 3.4. As previously stated, Fig. 2(c) is a neighborhood subgraph of vertex u_1 in Q. By using the neighborhood subgraph as the matching predicate, we obtain the neighborhood patterns of vertex u_1 in Q {(9,8,7,6,10)} in G, and we also can compute the candidate set with {9} in G for vertex u_1 in Q.

To speed up graph matching, we can index neighborhood patterns of vertices for fast retrieval of candidate vertices in G. As a result, sequential scan of all vertices in a large graph is avoided. In Algorithm 1, we present the overall procedure of graph matching by using the neighborhood patterns.

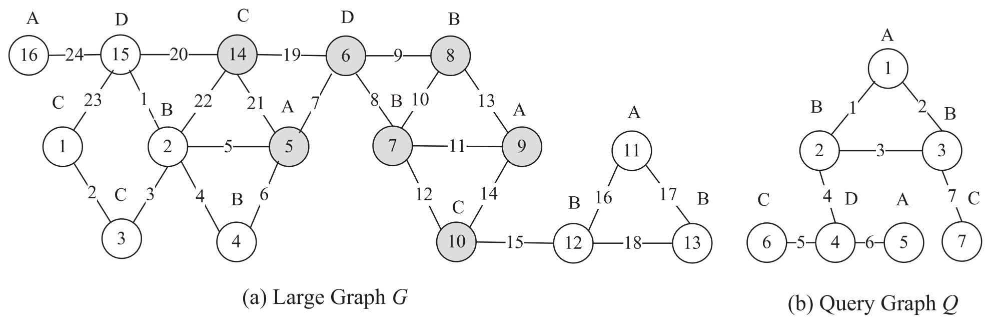


Fig. 1. A large graph G and a query graph Q.

Algorithm 1. GraphMatching

```
Input: A large graph $G$, A query graph $Q$, A nonnegative radius $r$;
Output: All feasible matchings $f_Q(G)$;
1:  for all vertex $u \in V(Q)$ do
2:    Generate neighborhood pattern $p_Q^r(u)$ for vertex $u$;
3:    $C(u) \leftarrow \{v | v \in V(G), \phi_u(v) = true\}$;
4:    Obtain all $C(p_Q^r(u)) \leftarrow \{p_G^r(v) | p_G^r(v) \subseteq G, v \in C(u)\}$;
5:  end for
6:  $P \leftarrow \varnothing$;
7:  Recursive_Match($P$);
8:
9:  void Recursive_Match($P$)
10: begin
11: Select an unvisited vertex $u \in V(Q)$ has been covered by
    $P$ with minimum cost and marked $u$ as visited
12: for all $p_G^r(v) \in C(p_Q^r(u))$ do
13:   if not Joinable($(p_Q^r(u), p_G^r(v)), P$) then
14:     continue;
15:   end if
16:   $P \leftarrow P \cup \{(p_Q^r(u), p_G^r(v))\}$;
17:   if $\forall e \in E(Q)$ has been covered by $P$ then
18:     Report $f_Q(G) \leftarrow \{p_G^r(v) | (p_Q^r(u), p_G^r(v)) \in P\}$;
19:   else
20:     Recursive_Match($P$);
21:   end if
22:   $P \leftarrow P - \{(p_Q^r(u), p_G^r(v))\}$;
23: end for
24: end
25:
26: boolean Joinable($(p_Q^r(u), p_G^r(v)), P$)
27: begin
28: for all $(p_Q^r(u'), p_G^r(v')) \in P$ do
29:   if $p_Q^r(u)$ and $p_Q^r(u')$ are joinable, while $p_G^r(v)$ and $p_G^r(v')$
      are not based on the same join predicate then
30:     return false;
31:   end if
32: end for
33: return true;
34: end
```

The algorithm consists of two phases. In the first phase (Lines 1–7), all candidates for each vertex of Q are retrieved by using the neighborhood pattern as the filtering condition. In addition, all neighborhood patterns of candidate vertices are computed. In the second phase (Lines 9–34), depth-first matching is performed to obtain all occurrences of Q in G with the search space of all

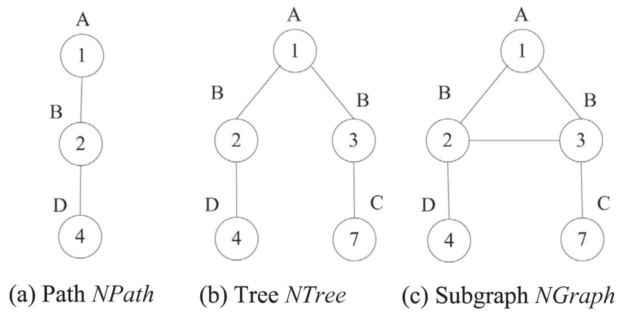


Fig. 2. Different neighborhood patterns of vertex u_1 in query Q.

$\prod_{i=1}^{L} |C(p_Q^r(u_i))|$, where L is the number of visited vertices during the matching space, and $C(p_Q^r(u_i))$ is the set of matched neighborhood patterns in G. Therefore, a small L and fewer matched neighborhood patterns will make the search space smaller. In Section 3.3, a detailed discussion on the selection of neighborhood patterns is given.

For each iteration of Recursive_Match, an unvisited node u is first selected on the basis of the cost model (Line 11). Then, for each neighborhood pattern $p_G^r(v)$, a join operation is performed with all previously visited patterns. If they are joinable, then a deep matching process is performed (Lines 16–21). Otherwise, joining operations on the next neighborhood pattern will continue (Lines 13–15). This procedure is repeated until all edges in Q have been covered by all of the discovered patterns in G. Consequently, injective mappings from Q to G appear, which is a result of matching (Lines 17–18). The possibility of the join operation is checked in the joinable function (Lines 26–34). If two neighborhood patterns in the query are joinable and their corresponding matched neighborhood patterns are not, then a failure of join process occurs.

3.3. Evaluation of neighborhood pattern

In this section, we evaluate the cost of performing the graph matching process shown in Algorithm 1. The cost of processing a graph matching query Q against G, denoted as C, can be modeled as the combination of vertex filtering cost for candidate pruning and pattern joining cost for query reconstruction:

$$C = C_f + C_j. \quad (5)$$

Here, C_f is the cost for filtering based on Eq. (1) in the candidate vertices generation phase, and C_j is the cost of pattern joining in the recursive matching phase.

For the first part, the vertex filtering cost is defined as $C_f = \frac{|V(Q)| \times |V(G)| \times C_{subiso}}{|\Sigma|}$, where C_{subiso} is the cost of subpattern isomorphism. For simplicity, we assume that the vertex labels of the large graph G are evenly distributed, such that for each vertex in Q, the number of vertices with the same label is $\frac{|V(G)|}{|\Sigma|}$.

For the second part, the pattern joining cost C_j is defined as $C_j = \prod_{i=1}^{L} |C(p_Q^r(u_i))| \times C_{subjoin}$, where $C_{subjoin}$ is an average cost of the join operations on the new neighborhood pattern with all discovered patterns.

The key issue to improve the matching performance is to minimize $C_{subiso}, L, C(p_Q^r(u_i))$, and $C_{subjoin}$. Intuitively, L and $C(p_Q^r(u_i))$ will be minimized if we employ neighborhood subgraphs as neighborhood patterns and index all neighborhood subgraphs of vertices in G. However, this measure is not feasible because subgraph isomorphism C_{subiso}, which is proved to be NP-complete, will incur a large computation overhead, whereas containment testing of trees requires polynomial time only [40]. That is, using the neighborhood subgraph will dramatically increase the total cost of C_j.

On the contrary, we can use the path to filter negative vertices and recover the original query graph. Although the cost of containment testing C_{subiso} is reduced, the numbers of L and $C(p_Q^r(u_i))$ will increase quickly. Moreover, neighborhood paths have weaker filtering and recovering abilities compared with trees and graphs. As a result, the search space of matching is dramatically increased. In particular, when the radius r is set to zero, which means neighborhood information is not used, an exhaustive search occurs. In this way, the search space of graph matching is $\prod_{i=1}^{|V(Q)|} |C(u_i))|$. Therefore, a pattern with both higher pruning power and quick containment testing is considered to be an ideal indexing object.

Based on the analysis mentioned above, we have drawn the following conclusion with respect to the structural neighborhood pattern: We should choose neighborhood trees to index the vertices in G to record the neighborhood information of the vertices. However, when using neighborhood trees as patterns, we still need to solve the following three problems: (1) how to make the neighborhood tree an effective neighborhood pattern, (2) how to quickly perform tree containment testing and tree join testing, and (3) how to select the initial neighborhood trees to recover the query. Details on techniques for resolving these problems will be addressed in subsequent sections.

4. Neighborhood tree index

In this section, we study an effective neighborhood pattern known as neighborhood tree. To quickly prune negative vertices, the level-wise signature that records the neighborhood edge information is first proposed. Then, the neighborhood tree-indexing schema based on the level-wise signature is presented. A tree-encoding scheme is proposed for fast tree containment testing and memory saving. In addition, index construction of neighborhood trees is described.

4.1. Neighborhood tree

Definition 6 (*d-Level Vertex Set*). Given $u \in V(G)$, a nonnegative distance d, the d-level vertex set, denoted as $V(u,d)$, is defined as

$$V(u,d) = \{v | hop(u,v) = d, v \in V(G)\}, \tag{6}$$

where $hop(u,v)$ is the number of hops away from u.

Definition 7 (*d-Distance Same Level Edge Set*). Given $u \in V(G)$, a nonnegative distance d, the d-distance same level edge set, denoted as $ES(u,d)$, is defined as

$$ES(u,d) = \{(v,w) | (v,w) \in E(G), v \in V(u,d), w \in V(u,d)\}. \tag{7}$$

Definition 8 (*d-Distance Adjacent Level Edge Set*). Given $u \in V(G)$, a nonnegative distance d, the d-distance adjacent level edge set, denoted as $EA(u,d)$, is defined as

$$EA(u,d) = \{(v,w) | (v,w) \in E(G), v \in V(u,d-1), w \in V(u,d)\}. \tag{8}$$

Definition 9 (*Level-wise Signature*). Given $u \in V(G)$ and radius r, the level-wise signature of u, denoted as $LS(u,r)$, is defined as

$$LS(u,r) = \{ES(u,d) | 1 \leqslant d \leqslant r\} \cup \{EA(u,d) | 1 \leqslant d \leqslant r\}. \tag{9}$$

It should be noted that $ES(u,d), EA(u,d)$, and $LS(u,r)$ are all multisets.

Example 4.1. We categorize the neighborhood vertices into rings according to the distances. In Fig. 3, neighborhood vertices arranged in the form of rings are presented for vertices v_9 and v_5 in G and vertex u_1 in Q. Their corresponding level-wise signatures are presented in Table 1. For vertex v_5 in G, the two-distance adjacent level edge set is $EA(v_5,2) = \{(B-C:1),(B-D:3),(C-D:1)\}$. This set indicates that edge $B-D$ occurs three times from vertices in the one-level set to those in the two-level set. The two-distance same-level edge set $ES(v_5,2) = \{(B-B:1)\}$.

Theorem 1. *Assume Q is subgraph isomorphic to G with mapping f; then, $\forall u \in V(Q)$, and we have $\forall r \geqslant 0, \bigcup_{E \in LS(u,r)} E \subseteq \bigcup_{E \in LS(f(u),r)} E$.*

Proof. According to our definition, $\bigcup_{E \in LS(u,r)} E$. That is, $\bigcup_{d \leqslant r} ES(u,d) \cup EA(u,d)$ represents all neighborhood edges within r-distance from u. Because $Q \subseteq G$, the neighborhood edge set of u should be contained by the neighborhood edge set of $f(u)$. Therefore, we have $\forall r \geqslant 0, \bigcup_{E \in LS(u,r)} E \subseteq \bigcup_{E \in LS(f(u),r)} E$. □

This pruning technique using neighborhood information is similar to the method of neighborhood signature (NS)-containment presented in SPath. Although the NS-containment method employs the profiles of neighborhood vertices as the pruning condition, we further employ level-wise edges that have more powerful pruning abilities.

Example 4.2. When setting $r = 2$, for vertex u_1 shown in Fig. 3(c) and its level-wise signature presented in Table 1, we have $\bigcup_{E \in LS(u_1,2)} E = \{(A-B:2),(B-B:1),(B-C:1),(B-D:1)\}$. For vertex v_9 shown in Fig. 3(a), we have $\bigcup_{E \in LS(v_9,2)} E = \{(A-B:2),(A-C:1),(B-B:1),(B-C:2),(B-D:2)\}$. From Theorem 1, we quickly compute that $\bigcup_{E \in LS(u_1,2)} \subseteq \bigcup_{E \in LS(v_9,2)}$ and verify that it is a candidate for u_1. Accordingly, v_5 is also a candidate for u_1. Although vertices v_{11} and v_{16} have the same labels as u_1, they are not candidates. For vertex v_{11}, we have $\bigcup_{E \in LS(u_1,2)} \nsubseteq \bigcup_{E \in LS(v_{11},2)}$. The same is true for vertex v_{16}. By using this filtering approach, the search space $\prod_{i=1}^{6} |C(u_i)|$ is dramatically reduced from 18,432 to 432 for query Q against graph G.

However, the level-wise signature method is still not powerful enough to prune some false positive candidates. We know that SPath uses a path-based graph-indexing mechanism for large networks [17]. However, some edge information will be lost after transforming a graph into several paths in the SPath method. We take Fig. 1 as an example to illustrate that SPath may result in false positives. In Fig. 1(b), three paths are given in query graph Q: $v_1(A) - v_2(B) - v_4(D) - v_6(C)$, $v_1(A) - v_2(B) - v_4(D) - v_5(A)$, and $v_1(A) - v_3(B) - v_7(C)$. It should be noted that these numbers are

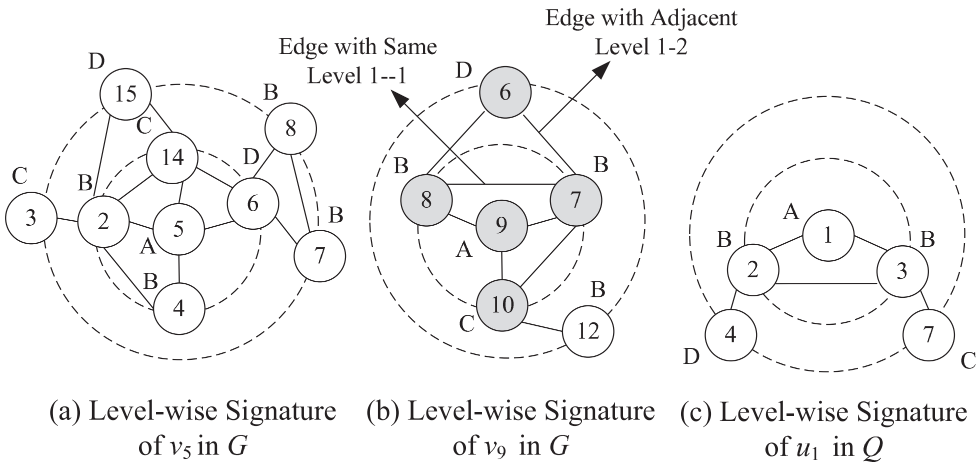

Edge with Same Level 1--1
Edge with Adjacent Level 1-2
(a) Level-wise Signature of v_5 in G
(b) Level-wise Signature of v_9 in G
(c) Level-wise Signature of u_1 in Q

Fig. 3. Neighborhood vertices arranged in the form of rings.

Table 1
Level-wise signatures for three different vertices with label A ($r = 2$).

Level	Neighborhood edges (v_5 in G)	Neighborhood edges (v_9 in G)	Neighborhood edges (u_1 in Q)
0–1	{(A-B:2), (A-C:1), (A-D:1)}	{(A-B:2), (A-C:1)}	{(A-B:2)}
1–1	{(B-B:1), (B-C:1), (C-D:1)}	{(B-B:1), (B-C:1)}	{(B-B:1)}
1–2	{(B-C:1), (B-D:3), (C-D:1)}	{(B-D:2), (B-C:1)}	{(B-C:1), (B-D;1)}
2–2	{(B-B:1)}	{}	{}

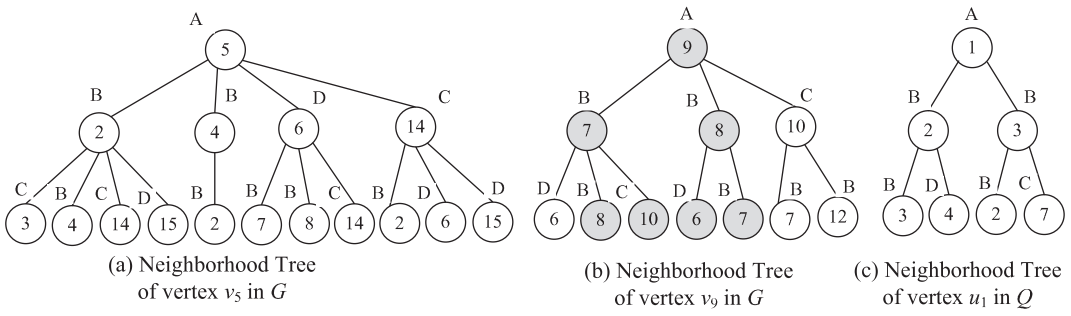

(a) Neighborhood Tree of vertex v_5 in G
(b) Neighborhood Tree of vertex v_9 in G
(c) Neighborhood Tree of vertex u_1 in Q

Fig. 4. Neighborhood trees for different vertices with $r = 2$.

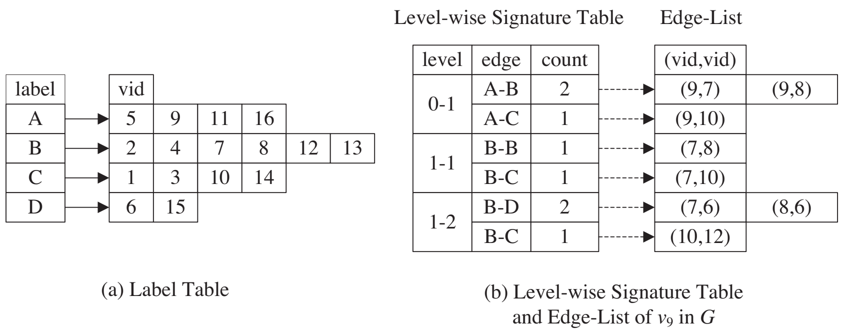


Fig. 5. Neighborhood tree index.

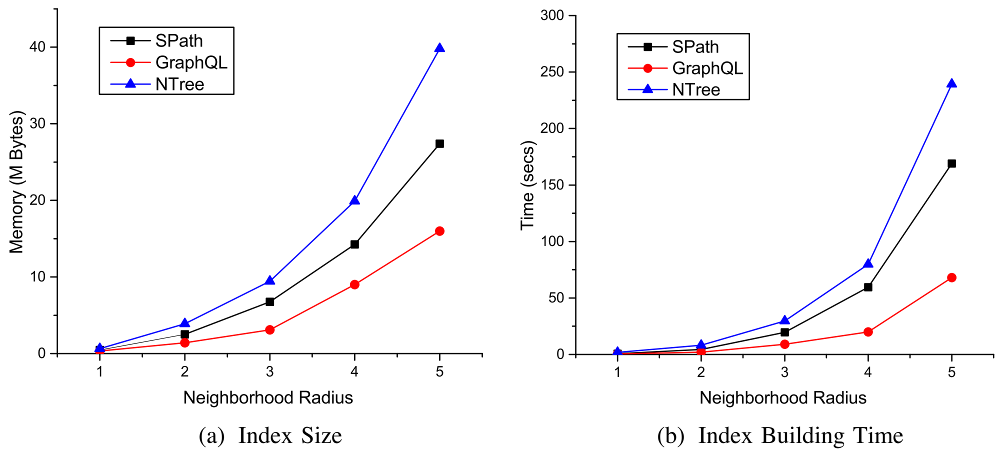


Fig. 6. Index construction for real dataset.

Table 2
Dataset and query set characteristics of the yeast protein interaction network.

Data (query)	Set size	Average vertex number	Average edge number	Average degree	Memory store (KB)
Real dataset	1	5657	37,221	13	514
Path query set	9000	6	5	1.7	2574
Clique query set	6000	4.5	9.5	4.2	2352
Normal query set	9000	6	11.7	3.9	3468

vertex identifiers in graph Q, whereas the uppercase letters inside the parentheses represent vertex labels in path representation. After conducting the query in graph G by using the above condition, we have three results:

1. Path 1: $v_5(A) - v_2(B) - v_{15}(D) - v_1(C)$,
   Path 2: $v_5(A) - v_2(B) - v_{15}(D) - v_{16}(A)$,
   Path 3: $v_5(A) - v_2(B) - v_3(C)$;
2. Path 1: $v_9(A) - v_7(B) - v_6(D) - v_{14}(C)$,
   Path 2: $v_9(A) - v_7(B) - v_6(D) - v_5(A)$,
   Path 3: $v_9(A) - v_7(B) - v_{10}(C)$;
3. Path 1: $v_9(A) - v_8(B) - v_6(D) - v_{14}(C)$,
   Path 2: $v_9(A) - v_8(B) - v_6(D) - v_5(A)$,
   Path 3: $v_9(A) - v_7(B) - v_{10}(C)$.

It should be noted that these numbers are vertex identifiers in graph G, whereas the uppercase letters inside the parentheses represent vertex labels in path representation. According to Theorem 1, both vertices v_5 and v_9 are considered as candidate mapping vertices for u_1. Obviously, the No. 3 result is the correct query result for graph Q, and the others are false positive results. However, by using a more powerful pruning technique such as neighborhood tree, we can further discard these two false positive results.

Definition 10 (*Neighborhood Tree*). Given graph G, vertex v, and radius r, a neighborhood tree $t^r_G(v)$ of G is consists of all of the longest paths starting from v with distances of no more than r. All edges in the neighborhood are denoted as $E(t^r_G(v))$.

Theorem 2. *Neighborhood tree is an effective neighborhood pattern. That is, given large graph G, query graph Q, and radius r, if Q is a subgraph isomorphic to G with mapping f, then $\forall u \in V(Q)$, $t^r_Q(u) \subseteq t^r_G(f(u))$.*

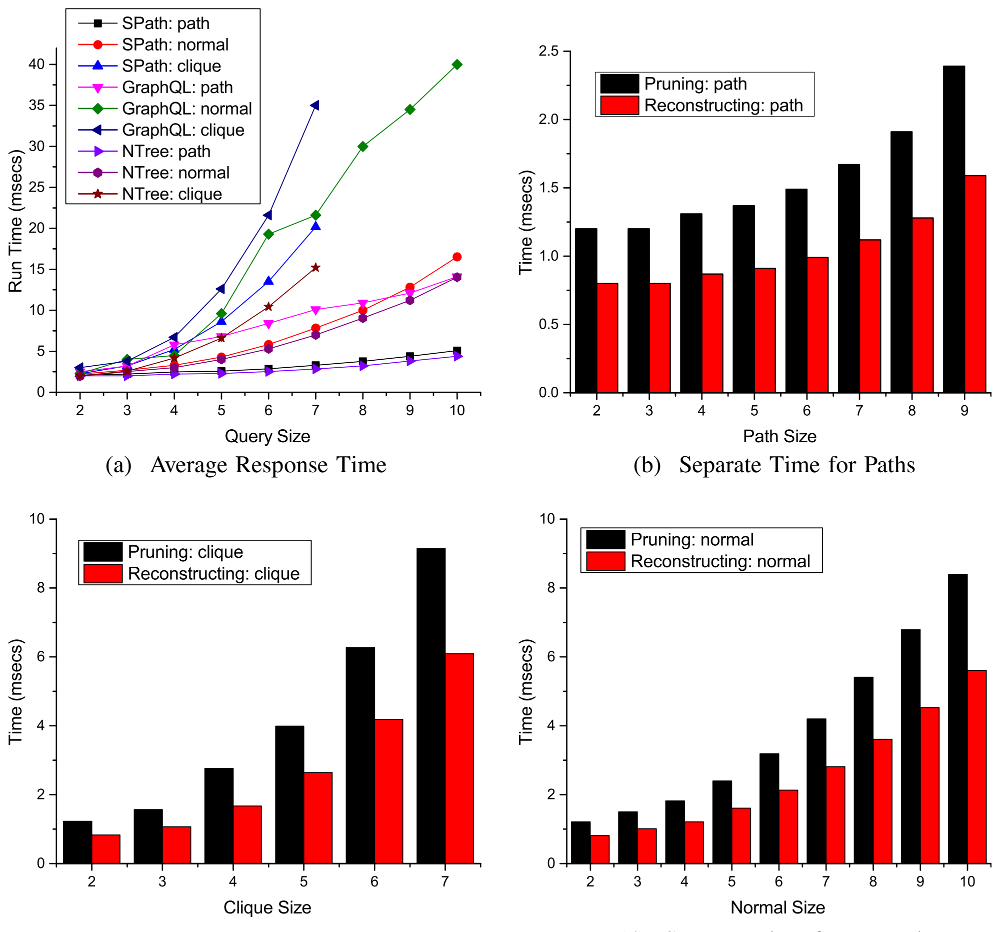
Proof. Given an arbitrary vertex $u_1 \in V(Q)$, suppose there is a path $(u_1, \ldots, u_k) \in t^r_Q(u_1)$ starting from u_1. Because $Q \subseteq G$, there must exists a series of vertices $f(u_1), \ldots, f(u_k)$ mapped by $u_1, \ldots, u_k$. Then, a path $(f(u_1), \ldots, f(u_k)) \in t^r_G(f(u))$ is found. This indicates that every path in $t^r_Q(u_1)$ has a corresponding mapping path in $t^r_G(f(u))$, and $t^r_Q(u_1) \subseteq t^r_G(f(u_1))$ satisfies the condition. Therefore, neighborhood tree is considered to be an effective neighborhood pattern. □

Example 4.3. Fig. 4 shows three different neighborhood trees for all vertices with label A. Because the neighborhood tree of v_5 does not contain a substructure of the neighborhood tree of u_1, v_5 should be pruned. Vertex v_9 is a candidate of u_1 because the substructure of its neighborhood tree colored in gray matches the neighborhood tree of u_1. In this way, we reduce the candidate set of u_1 to $\{v_9\}$ and further avoid false positive results.

Therefore, compared with the SPath method, our novel neighborhood tree-based approach works much better, particularly in avoiding more false positive results than that with the SPath method. This result occurred because unlike the path-based graph indexing mechanism, fewer pieces of structural information will be missing when transforming a graph into a neighborhood tree. In the following paragraphs, we describe several details in implementing the proposed neighborhood tree-based approach.

To solve the unordered tree containment problem, we first normalize the neighborhood tree into a unique depth-first canonical form as defined in previous research [41], and we then represent the canonical tree as a string for testing containment. Many ordered trees can be driven according to the ordering scheme of the vertices. From these ordered trees, we can uniquely select one as the canonical form to represent the corresponding unordered tree.

We adopt a string encoding scheme to represent the trees, which is more space-efficient and more easily manipulated.



(a) Average Response Time (b) Separate Time for Paths

(c) Separate Time for Cliques (d) Separate Time for Normals

Fig. 7. Query matching time for real dataset.

Assume that the string representation of tree T is denoted as S_T. To generate the string encoding, a depth-first traversal starting from the root is performed. When a vertex is visited during the traversal, the vertex label is appended to the end of the string. When a backtracking occurs from a child to its parent, a distinguished label ("#" here) not existing in the tree vertex label set needs to be appended to the string. The string encoding of tree T can differ. From all the tree strings encoded for tree T, the form of tree T with the minimal string value is considered as the canonical form of T, where the distinguished label "#" is defined as the largest label. We can construct the depth-first canonical form for an unordered tree in $O(cklogk)$ time by using a tree isomorphism algorithm given in previous research [42], where k is the number of vertices and c is the maximal degree of vertices in the tree.

Example 4.4. For the neighborhood trees in Fig. 4(b) and (c), the corresponding canonical trees are "A B B # C # D # # B B # D # # C B # B # #" and "A B B # C # # B B # D # #", respectively.

By taking advantage of the tree encoding scheme and canonical form, we apply the algorithm presented in previous research [40] to obtain all occurrences of neighborhood trees in the large graph G. The cost of tree containment testing will be minimal because we have pruned most of the negative vertices by using level-wise signatures, and the computing time of the tree containment testing is a polynomial. All occurrences of trees will be further used in the joining process.

4.2. Index construction

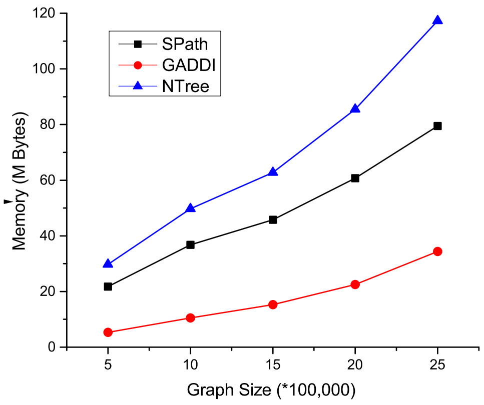
For actual implementation of the neighborhood tree index, we decomposed it into three components:

(1) **Label Table**: An inverted label index, in which the key is a label and value is the identifier list of vertices with that label.
(2) **Level-wise Signature Table**: A table recording the neighborhood edges of each vertex according to different levels. For each edge, only the number of occurrences is maintained.
(3) **Edge-List**: An edge list that keeps track of vertex identifiers for each edge.

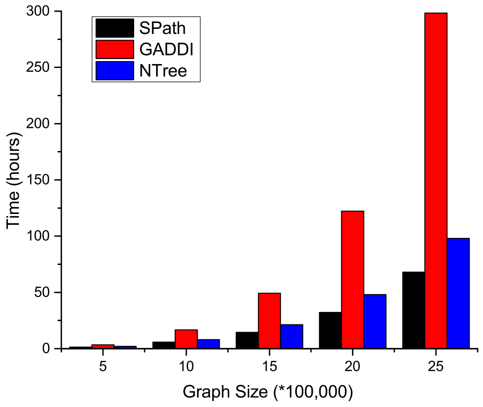
For the neighborhood tree signature, we can rebuild from the level-wise signature table and edge-list such that we can save a substantial amount of indexing space. Furthermore, on the basis of the neighborhood tree index, we can even rebuild the original graph because all edges in the graph are recorded.

Example 4.5. In Fig. 5(a), we present the label table for graph G; Fig. 5(b) shows the level-wise signature table of v_9 and the edge list.

When constructing the neighborhood tree index, each vertex should be visited, and its level-wise signature needs to be built. To accomplish this, a breadth-first search is required to visit edges



(a) Index Size



(b) Index Building Time

Fig. 8. Index construction for synthetic dataset.

within the distance of r. Moreover, the label table and edge-list can also be built. Assume the maximal degree of vertices is deg, then the time complexity to build a level-wise signature for a vertex is $\sum_{i=0}^{r}(deg^i)$. Therefore, the time cost for index construction is $O(|V(G)| \times \sum_{i=0}^{r}(deg^i))$, with $O(|V(G)| \times |E(G)|)$ as the worst case. Regarding space complexity, the edge-list takes up most of the space. For each vertex, $\sum_{i=0}^{r}(deg^i)$ number of edges is required to record the edge-list. Therefore, the total space cost is $O(|V| \times \sum_{i=0}^{r}(deg^i))$.

5. Query reconstruction

As stated in the pattern-based matching algorithm, when performing in each iteration of Recursive_Match, a pattern-based recovery is employed to reconstruct the query. In this way, we transform the matching process from vertex-at-a-time to pattern-at-a-time. In addition to pruning negative vertices, the neighborhood tree is also leveraged to reconstruct the query. In this section, we present the method for choosing the smallest of the neighborhood trees to cover the query in addition to the method for joining them.

5.1. Neighborhood tree join

Definition 11 (*Joining of Neighborhood Trees*). Given a neighborhood tree $t_G^r(u)$ of u and a neighborhood tree $t_G^r(v)$ of v, the joining of the two trees is defined as $E(t_G^r(u)) \cup E(t_G^r(v))$ if they have at least one common edge, such as $E(t_G^r(u)) \cap E(t_G^r(v)) \neq \emptyset$. The joining is denoted as $t_G^r(u) \bowtie t_G^r(v)$. The *join-predicates* are defined as the common edge set of $E(t_G^r(u)) \cap E(t_G^r(v))$.

If two neighborhood trees in graph G are *joinable*, they must satisfy the join-predicates as the corresponding neighborhood trees in Q, such as $\forall e \in E(t_Q^r(u)) \cap E(t_Q^r(v))$. Then, we have $f(e) \in E(t_G^r(f(u))) \cap E(t_G^r(f(v)))$. To accelerate the join process, edge identifiers are recorded to distinguish different edges and are also used for set operations.

Example 5.1. Taking u_1 and u_4 in query Q for example, the join-predicates of their neighborhood trees are the edge set $\{1, 3, 4\}$. For their matched candidates v_9 and v_6 in G, the join-predicates are $\{8, 9, 10, 11, 12, 13\}$. For edges, we have $f(1) = 13, f(3) = 10$, and $f(4) = 9$, which all can be found in the set $\{8, 9, 10, 11, 12, 13\}$. Thus, neighborhood trees of v_9 and v_6 are joinable. In this way, we can quickly recover the query to make use of trees.

5.2. Optimization of search order

In this section, we consider the selection strategy of neighborhood trees from query Q as well as their selecting order. To match a query in the large graph, only a small set of neighborhood trees that covers all of the edges in the query are required. However, it is a set-cover problem to reconstruct the query by using the smallest set of neighborhood trees, which is proved to be NP-complete. Therefore, as the methods proposed in previous research [16,17], a simple greedy approach is adopted for selecting neighborhood trees in each recursive joining process. To decide which neighborhood tree is selected in each iteration, we estimate the cost of each join and choose the neighborhood tree with the minimal cost.

Suppose when performing query matching, L neighborhood trees are selected for nodes $u_1, \ldots, u_L$ in order, denoted as $t_Q^r(u_1), \ldots, t_Q^r(u_L)$, respectively. For simplicity, we use t_i to represent $t_Q^r(u_i), 1 \leqslant i \leqslant L$. The product of cardinalities of matched neighborhood trees in G is considered as the cost evaluation of a joining process. Then, the cost of a join t_i with all visited neighborhood trees is formulated as

$$Cost(t_i) = Card(i-1) \times |C(t_i)|, \tag{10}$$

where $Card(i-1)$ is the number of all matched neighborhood trees in G for nodes from u_1 to u_{i-1}. $Card(i-1)$ can be defined as

$$Card(i-1) = Cost(t_i) \times \alpha_{i-1}, \tag{11}$$

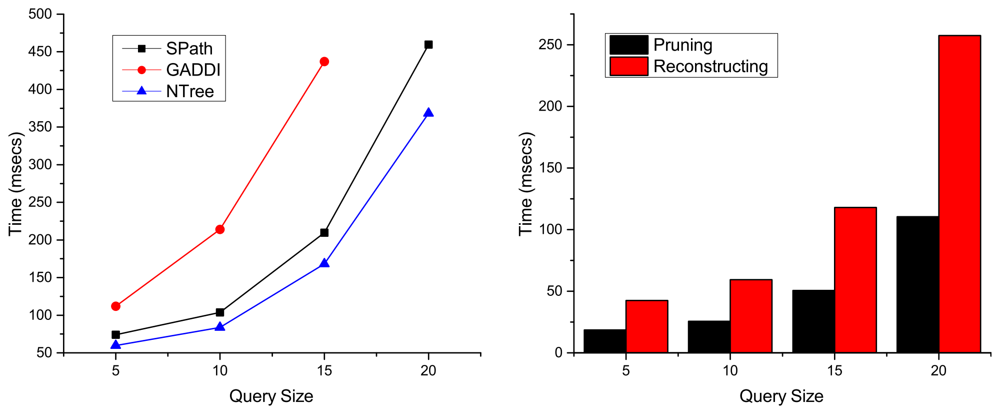
where $\alpha_{i-1} \in [0, 1]$ is a reduction factor. The total join cost of query Q can be formulated as

$$Cost(Q) = \sum_{i=1}^{L} Cost(t_i). \tag{12}$$

However, it is not easy to compute the total joining cost and obtain the joining order with the smallest cost. It should be noted that if we can reduce the joining count L and the result count of each join $Card(i)$, we can also minimize $Cost(Q)$. To achieve this goal, we find a neighborhood tree in Q that has fewer common edges with visited neighborhood trees in Q; fewer matched neighborhood trees in G may be a good pattern to be chosen in each recursion. In this way, we can not only cover the query quickly but also reduce the number of joining results. The selectivity of a neighborhood tree can be defined as

$$selectivity(t_i) = \frac{|E(t_Q^r(u_i)) - \cup_{k=1}^{k=i-1}(E(t_Q^r(u_k)))|}{|E(Q)| \times C(t_Q^r(u_i))}, \tag{13}$$

where we divide the probability of non-occurred edges of t_i in Q by the number of matched neighborhood trees for t_i in G. Our query



(a) Average Response Time

(b) Separate Time for Normals

Fig. 9. Query matching time for synthetic dataset.

optimizer always chooses a neighborhood tree with the largest selectivity value in a greedy approach. This also indicates that a new vertex with the longest distance from all visited vertices may have higher selectivity.

6. Experiments

In this section, we compare the performance of our proposed approach, NTree, with other state-of-the-art methods on both real and synthetic datasets. All algorithms were implemented in Java 7 and run on a 2.0 GHz Pentium dual machine with 8 GB memory running a Ubuntu operating system. When running the experiments, we set the following parameters for the Java virtual machine: Xms = 512 MB and Xmx = 2048 MB. The neighborhood radius r was set to 4 for both SPath and NTree if not specified explicitly.

6.1. Biological network

We adopted the same real dataset as that used in SPath, which is a yeast protein interaction network given by the Munich Information Center for Protein Sequences (MIPS) [43]. Six available datasets were downloaded and combined to construct a large graph with 5,657 vertices and 37,221 edges. To obtain a meaningfully large graph and corresponding queries, we added Gene Ontology (GO)[1] information as vertex labels to the proteins. The GO is a hierarchy of categories describing cellular components, biological processes, and molecular functions of genes and their products (proteins). Each GO term is a vertex in the hierarchy and has one or more parent GO terms; each protein has one or more GO terms. The original GO terms in the yeast protein interaction network contain 6,289 distinct labels. If we use the original GO terms as vertex labels, more than 2,700 distinct labels exist in the network. To further reduce the number of labels, high-level ancestors were used to relax the GO terms, which consist of only 136 distinct labels.

For the queries, we generated three different types of graphs: cliques, paths, and normal graphs. As stated in previous research [16], the cliques may correspond to protein complexes, whereas the paths may correspond to transcriptional or signaling pathways. The normal graphs are considered as general queries in the middle of two extremes of cliques and paths. For each normal graph and path, we generated 1000 queries varying in size from 2 to 10. For cliques, 1000 queries were generated from only size 2 to 7 because cliques with sizes greater than 7 have no answers. In Table 2, we present the characteristics of datasets and query sets. If a query had an excessive number of matchings in the network (more than 1000 answers), the matching process was terminated after 1000 answers were obtained. It should be noted that the queries returning no answers were not counted in the statistics. (SPath and GraphQL use the same statistical strategy.)

In Fig. 6, we evaluate the index construction cost for SPath, GraphQL, and NTree by using the yeast protein interaction network. Fig. 6(a) illustrates the space requirement by varying the neighborhood radius r from 1 to 5. Compared with SPath and GraphQL methods, NTree required more memory space to record neighborhood information for each vertex because it keeps track of the neighborhood edges; SPath and GraphQL record only neighborhood vertices. NTree can use these neighborhood edges to reconstruct the neighborhood trees and even the original graph. With an increase in r from 1 to 5, NTree grew linearly, which is similar to SPath and GraphQL. The memory used for NTree was less than 40 MB even when $r = 5$, which indicates that all neighborhood information is maintained when performing query matching. Fig. 6(b) illustrates the index construction time for these three approaches. NTree required 30% more time than that by SPath and three times that by GraphQL. However, even when $r = 5$, the running time of index construction for NTree was within 250 s.

In Fig. 7(a), we present the average response time of the three different types of queries for NTree, SPath, and GraphQL. For path queries, the matching performances of NTree and SPath were close, whereas the elapsed time of GraphQL was three times that of the NTree. Because path queries do not contain complex relationships, the pruning efficiency of both NTree and SPath approaches was similar. However, the GraphQL approach was less effective than the other two approaches. The joining processing procedures of these three approaches were also similar to each other. For clique queries, NTree outperformed SPath and GraphQL up to 30% and 50%, respectively. Compared with the pruning methods used in SPath and GraphQL approaches, which only filter vertices according to the profile of neighborhood vertices, NTree leveraged neighborhood trees, which contain more structural information. This result indicates that NTree has a more powerful pruning ability. Moreover, NTree decomposed the query into trees, which can reconstruct the query efficiently to result in the requirement of fewer joining processes when performing recursive matching. For normal queries, NTree also presented significant improvements in performance.

In Fig. 7(b)–(d), the separate steps of average processing time of NTree are shown respectively by varying the query sizes. The separate steps mainly included a vertices **pruning** process to obtain all candidates as well as matched neighborhood trees and a query

[1] http://www.yeastgenome.org/.

reconstruction process for recursive matching. As shown in the figures, the pruning process required the majority of time for query matching because the generation of all candidates and matched neighborhood trees in the network requires a substantial amount of time.

6.2. Synthetic graphs

We also evaluated the query matching performance on synthetic graphs. Synthetic datasets were generated by using the same method as that proposed in SPath. Graphs were generated on the basis of the Recursive Matrix model [44] following the power law in- and out-degree distributions. The parameters were set by the default values specified in this paper. When performing this process, the label table and level-wise signature table were maintained in the main memory to speed up the matching process. Five large networks with vertex numbers from 500,000 to 2,500,000 were generated, in which the number of edges was five times of that of vertices. The vertex labels were generated randomly from the label set with size $1\% \times |V(G)|$.

In Fig. 8, we present the evaluation of index construction for NTree, SPath, and GADDI [18] approaches. The GraphQL approach failed in all of these scenarios because of the huge quantity of vertices and edges. Therefore, we compared our proposed method NTree with the GADDI approach instead, which is a state-of-the-art approach for mining large-scale networks exclusively. As shown in Fig. 8(a), NTree scaled linearly with an increase in graph size. Therefore, NTree, SPath, and GADDI have good scalability in terms of index size. Although the running time increased rapidly when the graph size increased, the index construction time is still acceptable considering that only one index construction was required.

In Fig. 9, we further compare the matching performance of these three approaches on the synthetic graph G with $|V(G)| = 1,000,000$ and $|E(G)| = 5,000,000$. Queries were generated with various sizes of 5, 10, 15, and 20 by randomly extracting induced subgraphs from G. Similar to the previous generating method, for each size, 1,000 queries were generated, and the average response time was evaluated. As shown in Fig. 9(a), NTree outperformed SPath and GADDI in terms of efficiency up to 20% and 50%, respectively. As stated previously, this result is attributed to the improvement of pruning and reconstruction abilities. For the GADDI approach, we did not give the result when the query size was 20, because such a size failed to complete its query processing in a reasonable amount of time. In Fig. 9(b), we also illustrate the running time of separate steps for query matching. The time required for the reconstruction process was increased dramatically because the retrieval of edge-lists from the disk is needed for each joining process.

7. Conclusions

In this paper, we propose a pattern-based graph-matching algorithmic framework. We evaluate the vertex pruning and query reconstruction abilities of three structural patterns: paths, trees, and subgraphs. Trees are considered as the best patterns for indexing graphs. A new graph-indexing mechanism neighborhood tree is proposed by preserving the local structural information of vertices for candidate selection and query reconstruction. To recover the query, a tree joining strategy is presented. Furthermore, a cost evaluation model considering both the pruning and reconstruction abilities is also described. Through extensive experiments on real and synthetic graph datasets, we demonstrate that our NTree is a scalable and efficient graph-indexing technique and can outperform state-of-the-art methods such as SPath.

Acknowledgments

The work was supported in part by the National Key Technology R&D Program (Grant Nos. 2014BAH23F03 and 2014BAH28F05), the National Natural Science Foundation of China (Grant Nos. 61379034 and 60903038), the China Scholarship Council (Grant No. 201306320037), the Zhejiang Province Science and Technology Innovation Team on E-Commerce (Grant No. 2010R50041), Smart Industry Talent Base Project of Ningbo.

References

[1] A. Golovin, K. Henric, Chemical substructure search in sql, J. Chem. Inform. Model. 49 (1) (2009) 22–27.
[2] R. Wallace, Structure and dynamics of the 'protein folding code' inferred using tlusty's topological rate distortion approach, Biosystems 103 (1) (2011) 18–26.
[3] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, A. Ferro, A subgraph isomorphism algorithm and its application to biochemical data, BMC Bioinform. 14 (Suppl. 7) (2013) 1–13.
[4] D. Deutch, T. Milo, N. Polyzotis, Top-k queries over web applications, VLDB J. 22 (4) (2013) 519–542.
[5] W.J. Christmas, J. Kittler, M. Petrou, Structural matching in computer vision using probabilistic relaxation, IEEE Trans. Pattern Anal. Machine Intell. 17 (8) (1995) 749–764.
[6] N.V. Dokholyan, L. Li, F. Ding, E.I. Shakhnovich, Topological determinants of protein folding, Proc. National Acad. Sci. 99 (13) (2002) 8637–8641.
[7] X. Yan, P. Yu, J. Han, Graph indexing: a frequent structure-based approach, in: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, 2004, pp. 335–346.
[8] P. Zhao, J.X. Yu, P.S. Yu, Graph indexing: Tree + delta <= graph, in: Proceedings of the 33rd International Conference on Very Large Data Bases, 2007, pp. 938–949.
[9] S. Zhang, M. Hu, J. Yang, Treepi: a novel graph indexing method, in: Proceedings of the IEEE 23rd International Conference on Data Engineering, 2007, pp. 966–975.
[10] S. Seufert, A. Anand, S. Bedathur, G. Weikum, Ferrari: Flexible and efficient reachability range assignment for graph indexing, in: Proceedings of the IEEE 29th International Conference on Data Engineering (ICDE), 2013, pp. 1009–1020.
[11] J. Cheng, Y. Ke, W. Ng, A. Lu, Fg-index: towards verification-free query processing on graph databases, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, 2007, pp. 857–872.
[12] H. He, A.K. Singh, Closure-tree: an index structure for graph queries, in: Proceedings of the 22nd International Conference on Data Engineering, 2006, pp. 38–49.
[13] L. Zou, L. Chen, J.X. Yu, Y. Lu, A novel spectral coding in a large graph database, in: Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology, 2008, pp. 181–192.
[14] L. Zhu, Q.B. Song, Y.C. Guo, L. Du, X.Y. Zhu, G.T. Wang, A coding method for efficient subgraph querying on vertex- and edge-labeled graphs, PLoS ONE 9 (5) (2014) 1–18.
[15] D. Pal, P. Rao, A tool for fast indexing and querying of graphs, in: Proceedings of the 20th International Conference Companion on World Wide Web, 2011, pp. 241–244.
[16] H. He, A.K. Singh, Graphs-at-a-time: query language and access methods for graph databases, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, 2008, pp. 405–418.
[17] P. Zhao, J. Han, On graph query optimization in large networks, in: Proceedings of the VLDB Endowment, vol. 3, 2010, pp. 340–351.
[18] S.J. Zhang, S.R. Li, J. Yang, Gaddi: distance index based subgraph matching in biological networks, in: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, 2009, pp. 192–203.
[19] D. Eppstein, Subgraph isomorphism in planar graphs and related problems, J. Graph Algor. Appl. 3 (3) (1999) 1–27.
[20] J. Ullmann, An algorithm for subgraph isomorphism, J. ACM 23 (1) (1976) 31–42.
[21] B. Gallagher, Matching structure and sematincs: a survey on graph-based pattern matching, in: Proceedings of the Conference of the American Association for Artificial Intelligence, 2006, pp. 45–53.
[22] S. Ranu, M. Hoang, A. Singh, Answering top-k representative queries on graph databases, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 2014, pp. 1163–1174.
[23] D.Y. Yuan, P. Mitra, C.L. Giles, Mining and indexing graphs for supergraph search, Proc. VLDB Endowment 6 (10) (2013) 829–840.
[24] S. Zhang, J. Li, H. Gao, Z. Zou, A novel approach for efficient supergraph query processing on graph databases, in: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, 2009, pp. 204–215.
[25] W.S. Han, J. Lee, J.H. Lee, Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp. 337–348.

[26] M. Gupta, A. Mallya, S. Roy, J.H.D. Cho, J.W. Han, Local learning for mining outlier subgraphs from network datasets, in: Proceedings of the 2014 SIAM International Conference on Data Mining, 2014, pp. 73–81.
[27] R.D. Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, D. Shasha, Sing: subgraph search in non-homogeneous graphs, BMC Bioinform. 11 (1) (2010) 1–15.
[28] L. Zou, J.H. Mo, L. Chen, M.T. Özsu, D.Y. Zhao, gstore: answering sparql queries via subgraph matching, Proc. VLDB Endowment 4 (8) (2011) 482–493.
[29] X. Yan, P.S. Yu, J. Han, Substructure similarity search in graph databases, in: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, 2005, pp. 766–777.
[30] G.P. Zhu, X.M. Lin, K. Zhu, W.J. Zhang, J.X. Yu, Treespan: efficiently computing similarity all-matching, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012, pp. 529–540.
[31] Y. Tian, R.C. McEachin, C. Santos, D.J. States, J.M. Patel, Saga: a subgraph matching tool for biological graphs, Bioinformatics 23 (2) (2007) 232–239.
[32] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, S. Tao, Neighborhood based fast graph search in large networks, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, 2011, pp. 901–912.
[33] S.Q. Yang, Y.H. Wu, H. Sun, X.F. Yan, Schemaless and structureless graph querying, Proc. VLDB Endowment 7 (7) (2014) 565–576.
[34] W.G. Zheng, L. Zou, X. Lian, D. Wang, D.Y. Zhao, Graph similarity search with edit distance constraint in large graph databases, in: Proceedings of the 22nd ACM International Conference on Information & Knowledge Management, 2013, pp. 1595–1600.
[35] X. Zhao, C. Xiao, X.M. Lin, W. Wang, Y. Ishikawa, Efficient processing of graph similarity queries with edit distance constraints, VLDB J. 22 (6) (2013) 727–752.
[36] A. Khan, Y.H. Wu, C.C. Aggarwal, X.F. Yan, Nema: fast graph search with label similarity, Proc. VLDB Endowment 6 (3) (2013) 181–192.
[37] Z. Sun, H.Z. Wang, H.X. Wang, B. Shao, J.Z. Li, Efficient subgraph matching on billion node graphs, Proc. VLDB Endowment 5 (9) (2012) 788–799.
[38] H.H. Hung, S.S. Bhowmick, B.Q. Truong, B. Choi, S.G. Zhou, Quble: towards blending interactive visual subgraph search queries on large networks, VLDB J. 23 (3) (2014) 401–426.
[39] J.W. Huang, K. Venkatraman, D.J. Abadi, Query optimization of distributed pattern matching, in: Proceedings of the IEEE 30th International Conference on Data Engineering (ICDE), 2014, pp. 64–75.
[40] R. Shamir, D. Tsur, Faster subtree isomorphism, J. Algor. 33 (2) (1999) 267–280.
[41] Y. Chi, Y. Yang, R.R. Muntz, Canonical forms for labelled trees and their applications in frequent subtree mining, Knowl. Inform. Syst. 8 (2) (2005) 203–234.
[42] A.V. Aho, J.E. Hopcroft, J.E. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company, United States, 1974.
[43] A. Saurabh, D.K. Oliver, D.G. Francis, P.R. Frederick, Predicting protein complex membership using probabilistic network reliability, Genome Res. 14 (6) (2004) 1170–1175.
[44] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-mat: a recursive model for graph mining, in: Proceedings of SIAM International Conference on Data Mining, 2004, pp. 442–446.