Applied Mathematics & Information Sciences
An International Journal

Measuring Similarity between Graphs Based on the Levenshtein Distance

Bin Cao, Ying Li and Jianwei Yin

College of Computer Science and Technology, Zhejiang University, Hangzhou, China 310027

Received: 20 Oct. 2012, Revised: 29 Nov. 2012, Accepted: 11 Dec. 2012
Published online: 1 Feb. 2013

Abstract: Graph data has been commonly used and widely researched both in academia and industry for many applications. And measuring similarity between graphs (i.e., graph matching) is the essential step for graph searching, pattern recognition and machine vision. At present, the most widely used approach to address the graph matching problem is graph edit distance (GED). However, the computation complexity of GED is expensive and it takes unacceptable time when the graph becomes larger. Generally, graph could be canonical labeled by some sort of strings and we use the depth-first search (DFS) code as our canonical labeling system. Based on DFS codes, combining the Levenshtein distance (i.e., string edit distance, SED), we proposed a novel method for similarity measurement of graphs. Processing and calculating the distance between two DFS codes, we turned the graph matching problem into string matching, which gains great improvement on the matching performance. The experimental results prove its usefulness.

Keywords: Graph matching, similarity, depth-first search (DFS), Levenshtein distance

1. Introduction

As one of the most powerful structures, graphs can contain richer information than other data structures and they have been widely investigated and applied in a broad range of areas. Especially, graphs which are labeled and/or attributed can be used to abstract and model many complicated relations among data. When using graphs for representation, vertices usually represent regions (or features) of the objects and edges between them represent the relations between region. For example, World Wide Web (WWW) can be viewed as a graph in which vertices correspond to static pages and edges correspond to links between pages [1]. In business process, the labeled graphs are commonly used to model the real business operations and the business activities are represented by the vertices of the graphs.

Since many problems could be solved more easily based on graphs, people have collected vast amounts of graph data and established graph database for different purposes. Meanwhile, the academic communities have paid a lot of attentions on graph related researches. Among which, measuring the similarity between graphs is one of the hottest topics and it is the foundation for many other researches or applications. For example, to support scalable graph search over large graph databases in bioinformatics [2], chemical informatics [3], and even in business process management [4], it is essential to match the graphs by measuring their similarities.

Up to now, the most widely accepted method for graph similarity measurement is graph edit distance (GED) [5]. The basic idea of GED is to sum the cost of elementary 'error-correcting' operations: node substitution, node insertion/deletion, edge insertion/deletion. And the minimal cost taken over all operations is the edit distance between two graphs. Based on GED, a number of approaches have been proposed [6–9]. Unfortunately, the problem of GED is NP-hard in general and its main drawback is the exponential computational complexity in terms of the number of graph edit vertices [8]. Thus, Z.Zeng et al.[8] introduce a notion of so called star representation for graph structures and propose three novel methods to obtain lower and upper bounds of GED in polynomial time. However, their lower bound of computational complexity is in $O(n^3)$ which is still kind of expensive for computation involving a large amount of graphs. X. Yan et al. [9] propose a feature-based method for similarity search in graph structures. They use indexed features in graph database to filter graphs without performing pairwise similarity

* Corresponding author e-mail: cnliying@zju.edu.cn

computation. But they still turn to GED for measuring similarity when graph matching is needed.

In order to improve the efficiency of graph matching problem, in this paper we propose a novel method for measuring the similarity between two graphs. The start point of our method is the depth-first search code (DFS code)[10] and instead of GED measurements, we use Levenshtein [11] distance (i.e., string edit distance, SED) to measure the similarity between two graphs. The computation for SED is in $O(n^2)$ time which makes our method applicable in practice.

The rest of this paper is organized as follows. Section 2 will formally present some basic definitions for accurate description of graphs, DFS code, SED, and etc. Then the implementation details will be presented in Section 3. The experimental evaluations are studied in Section 4. Section 5 concludes the paper and presents some future work.

2. Basic Definitions

In our paper, we consider graphs with labeled nodes and edges. And we present some basic definitions as follows.

Definition 1(Labeled Graph). *A labeled graph is a tuple $G = (V, E, L_V, L_E, l)$, where V is a set of finite vertices and $E \subseteq V \times V$ is a set of edges.L_V and L_E denote the finite sets of vertex and edge labels. l is the mapping function for labels.*

From Definition 1, since the edge is denoted by two nodes, if there is an order between these two nodes then this is a directed graph, otherwise, an undirected graph. Besides, if $V(G_1) = V(G_2)$ and $E(G_1) = E(G_2)$, we consider graph G_1 and G_2 are the same. And G_1 is isomorphic to G_2 (i.e., $G_1 \cong G_2$) if they share the same structure.

Definition 2(Graph Isomorphism). *Let G and G' be two graphs. A graph isomorphism between G and G' is a bijective mapping $f : V(G) \to V(G')$ such that:*

$-\forall u \in V, (l(u) = l'(f(u)))$
$-\forall u, v \in V, ((u,v) \in E \Rightarrow (f(u), f(v)) \in E')$ *and*
$-\forall (u,v) \in E, (l(u,v) = l'(f(u), f(v)))$

As shown in Figure 1, G_1, G_2 and G_3 have the same number of nodes and edges. Besides, each edge in G_1, G_2 and G_3 is same since their corresponding start and end nodes are same. Through replacing the nodes in G_1, G_1 could be redrawn to G_2. Clearly, the difference among G_1, G_2 and G_3 is the way of labeling and drawing the graphs. That is to say, they have the same structure and they are merely different forms of a certain graph which is just a "4-circles" graph. From Figure 1, we can see that: (1) x_2 in G_2 corresponds to y_1 in G_3, (2) y_1 in G_2 corresponds to x_2 in G_3. Apparently, since the mapping between nodes in G_2 and G_3 is not unique, there are other mappings existed. And other drawings for G_2 could be found.

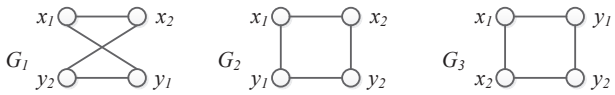


Figure 1 Three isomorphic graphs

In other words, if the topology of two graphs is same, then these two graphs are isomorphic. The isomorphism is very common in graph data. Under some circumstances, such as frequent subgraph mining, the isomorphic graphs should be pruned for the reason of efficiency. As for our work, since isomorphic graphs could be viewed as the same graphs, we needn't match all the graphs one by one. Instead, choosing one of the same graphs to measure is reasonable. Thus, before measuring the similarity, we have to examine the graph isomorphism.

In order to solve the isomorphism problem, we need to calculate the canonical labels of two graphs. The canonical label for a graph (denoted as $cl(G)$) is a unique code which is a sequence of bytes, characters or numbers. It is irrelative with the order of vertices and edges of the graph G and totally depends on the topology of G. If the canonical labels of two graphs are the same, then these graphs are isomorphic to each other. There are a few canonical labeling methods that have been applied, for example, concatenating rows or columns of the adjacency matrix of a graph. In our work, we introduce the depth-first search code (DFS code), which first mentioned in gSpan [10] algorithm, as the foundation of our canonical labeling system. Next, we present the necessary information of DFS code and more details refer to gSpan [10].

Depth-first search is well-known and popularly applied in graph algorithms and it consumes less memory than breadth-first search (BFS). When performing a depth-first search in a graph, a DFS tree would be constructed. The DFS trees of one graph maybe various, which is determined by the visiting order of the vertices in the graph. Thus, we can't examine the isomorphism of two graphs by DFS sequences. Adopting the DFS lexicographic order and the minimum DFS code as the canonical labeling can solve this problem. First of all, we present the definition of DFS subscripting as follows.

Definition 3(DFS Subscripting). *When building a DFS tree T, the depth-first discovery of the vertices forms a linear order. The subscripts are used to record this order, where $i < j$ means v_i is visited before v_j when the DFS is performed. G_T represents a graph G subscripted with T. T is called a DFS subscripting of G.*

We call v_0, the starting vertex in T, the root. The vertex v_n which visited last is called the rightmost vertex. The straight path from v_0 to v_n is called the rightmost path.

As shown in Figure 2, the vertex labels are X, Y and Z while the edges labels are a and b. The darkened edges in

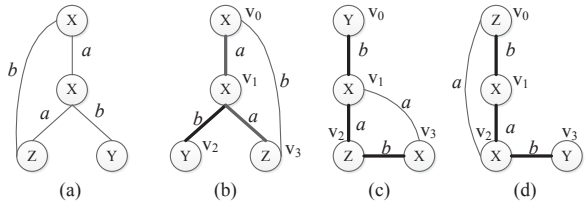


Figure 2 The sample of DFS subscripting

Figure 2(b) to Figure 2(d) represent three different DFS trees for the graph of Figure 2(a) and they generate three different subscriptings. The rightmost path for Figure 2(b) is (v_0, v_1, v_3) and (v_0, v_1, v_2, v_3) is for Figure 2(c) and Figure 2(d).

Definition 4(Rightmost Extension). *Given a graph G and its DFS tree T, we have:*

- *–Backward extension: a new edge can be added between the rightmost vertex and another vertex on the rightmost path.*
- *–Forward extension: a new vertex can be introduced and connected to a vertex on the rightmost path. Since both kinds of above extensions take place on the rightmost path, we call them rightmost extension.*

Taking Figure 2(c) as an example, since the edges already exist between v_1, v_2 and v_3, the backward extension candidates can be (v_3, v_0) and the forward extension candidates can be edges extending from v_0, v_1, v_2 or v_3, with a new vertex introduced. The all potential rightmost extensions of Figure 2(c) are shown in Figure 3. The dashed lines represent the extensions. Among which, Figure 3(a) and 3(b) both extend from the rightmost vertex (i.e., v_3) while Figure 3(c) to 3(d) are extend from other vertices on the rightmost path. Anyway, backward extension can only occur on the rightmost vertex and forward extension takes place on the vertex which belongs to the rightmost path.

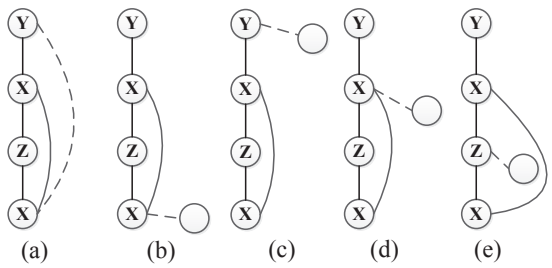


Figure 3 The rightmost extension for Figure 2(c)

As mentioned before, it is likely that one graph may have more than one DFS trees/subscriptings. In order to avoid the extension of the same graphs (i.e., isomorphic graphs), we have to choose one base subscripting and conduct rightmost extension on it.

Definition 5(DFS Code). *Given a DFS tree T for a graph G, based on rightmost extension, the subscripted graph G_T could be transformed to an edge sequence $e_i (i = 0, ..., |E| - 1)$. e_i is called a DFS code, denoted as $DFSCode(G, T)$.*

Based on Definition 5, there is a bijective mapping between a subscripted graph and a DFS code. Besides, since there are various edge sequences for a given graph G, we can build an order between these sequences and select the subscripting which generates the minimum sequence as the subscripting of G. This order could also be applied to DFS codes and we present it as follows.

Definition 6(DFS Lexicographic Order). *Let an edge be a 5-tuple: $(i, j, l_i, l_{(i,j)}, l_j)$, where li and l_j are the labels of v_i and v_j, respectively, and $l_{(i,j)}$ is the label of the edge connecting them. Given a vertex v, the edge order is that:*

- *–All of its backward edges should appear just before its forward edges.*
- *–If v does not have any forward edge, we put its backward edges after the forward edge, where v is the second vertex.*

Let the edge order take the first priority, the vertex label l_i take the second priority, the edge label $l_(i, j)$ take the third and the vertex label l_j take the fourth to determine the order of two edges. The ordering based on above rules is called DFS lexicographic order.

From Definition 6 it follows that, the complete sequence for Figure 2(c) is $(0,1),(1,2),(2,3),(3,1)$. The DFS codes for Figure 2(b) to 2(d) are shown in Table 1. We can see from Table 1 that the first edges of the DFS codes are $(0,1,X,a,X),(0,1,Y,b,X)$ and $(0,1,Z,b,X)$. Since they have the same subscript $(0,1)$ and no edge order exists between them, we can't use it to tell the difference among them. However, using the rest priorities of label information we have $(0,1,X,a,X) < (0,1,Y,b,X) < (0,1,Z,b,X)$. Therefore, $c_b < c_c < c_d$ is the order for the DFS codes listed in Table 1.

Table 1 DFS codes for Figure 2(b) to 2(d)

edge	c_b	c_c	c_d
e_0	$(0,1,X,a,X)$	$(0,1,Y,b,X)$	$(0,1,Z,b,X)$
e_1	$(1,2,X,b,Y)$	$(1,2,X,a,Z)$	$(1,2,X,a,X)$
e_2	$(1,3,X,a,Z)$	$(2,3,Z,b,X)$	$(2,0,X,a,Z)$
e_3	$(3,0,Z,b,X)$	$(3,1,X,a,X)$	$(2,3,X,b,Y)$

Definition 7(Minimum DFS Code). *Given a graph G, C(G) = {(DFSCode(G, T)) | ∀T, T is a DFS tree for G}, based on DFS lexicographic order, the minimum element of C(G) is called minimum DFS code, denoted as minDFSCode(G).*

According to Definition 7, the minimum DFS code of Figure 2(a) is c_b shown in Table 1. What is more, we can infer the following important relationship between the minimum DFS codes and isomorphic graphs.

*Property 1.*Given two graphs G and G', we have:

$$G_1 \cong G_2 \Leftrightarrow minDFSCode(G) = minDFSCode(G')$$

Proof:"⇒": Since G is isomorphic to G', according to Definition 1, G and G' is one-to-one mapped under some certain function: $f : V(G) \rightarrow V(G')$. Thus, based on the mapping between $E(G)$ and $E(G')$ and Definition 5, we can infer the mapping of $DFSCode(G,T) \rightarrow DFSCode(G',T)$. Naturally, we have $minDFSCode(G) \rightarrow minDFSCode(G')$. The proof is similar to "⇐".

On the basis of above discussions, we can use the minimum DFS code as the canonical label of one graph. At the end of this section, we present the definition of the Levenshtein distance (i.e., string edit distance, SED).

Definition 8(String Edit Distance, SED). *Given two strings x and y. The string edit distance of x and y, denoted as SED(x, y), is the minimum number of insertions, deletions and substitutions to transform x into y.*

Since the canonical label of a graph is the minimum DFS code which could be viewed as a string, we can measure the similarity between two graphs by conducting string edit distance (SED) calculation on their minimum DFS codes. Thus, we turn the graph matching problem into string matching problem which is much easier to solve. What is more, the string edit distance guarantees the efficiency of our work.

Based on the definitions introduced in this section, we present the implementation details in the following section.

3. Implementation

In this section, we discuss the implementation details of measuring similarity between graphs based on the DFS code mentioned in Section 2. Note that, we view the graph which is used to match against the graph database as the source graph.

According to the real requirements of different application scenarios, we can divide our implementation into two main phases which are preprocessing and matching. Preprocessing phase is performed offline while matching is online. Usually, people pay much attention to matching phase since it has direct influences on user experiences driven by efficient performance. Firstly, we present the pseudo code of preprocessing phase in Algorithm 1.

Algorithm 1 The algorithm for preprocessing phase

Input: Graph database (GD)
Output: The minimum DFS codes for GD: M_1; the orders of node labels and edge labels: N, E

```
1:  Initialize two map structures: M1 and M2
2:  N ← get order of node labels in GD
3:  E ← get order of edge labels in GD
4:  for each graph G in GD do
5:      ID ← get the ID of G
6:      code ← get minDFSCode(G) by N and E
7:      M1.put(ID, code)
8:  end for
9:  for each record r in M1 do
10:     add ID of r to the graph ID set: ID_Set
11:     for each record r' in M1 − {(ID, code)} do
12:         if the code in r is same with that in r' then
13:             add ID of r' to the ID_Set
14:             M2.put(code, ID_Set)
15:         end if
16:     end for
17: end for
18: for each record r = (code, ID_Set) in M2 do
19:     code ← extract two node labels of one edge in their
        appearing order from the code
20: end for
```

As shown in Algorithm 1, the input for this phase is the graph database which may contain large number of graphs. And this phase generates three outputs: the minimum DFS codes for all the graphs in the graph database; the orders of node labels and edge labels. At first, we parse each graph in the graph database and produce two orders of all the node labels and edge labels existing in the graph database (line 2 and 3). These orders are used for constructing the DFS code of the graphs in the following steps. By iterating the graphs in graph database (line 4-8), we get the ID and the minimum DFS code of each graph and put them in a map. Then, we merge the same codes in the graph database and using an inverted key-value pair (i.e., the minimum DFS code is the key and the value is graph IDs) to represent the DFS codes of the graph database (line 9-17). Thus, we put the same or isomorphic graphs together and we only select one of them for similarity measurement, which is efficient for matching. In fact, before calculating the SED, we firstly preprocess the minimum DFS code for simplicity by extracting the labels of two vertices of an edge in their appearing order (line 18-20). For example, the minimum DFS code of Figure 2(a) is $(0,1,X,a,X)(1,2,X,b,Y)(1,3,X,a,Z)(3,0,Z,b,X)$ which would be extracted to the string "XXXYXZZX".

Besides, in order to correctly extract the DFS code of the source graph and conduct similarity measurement between graphs in online matching, we must guarantee the consistency of the orders of node labels and edge labels in the whole matching procedure. Therefore, we record these orders and output them for matching phase.

Then, we present the online matching phase in Algorithm 2. Besides the results of the offline preprocessing phase, we add the source graph as another input. In this phase, we output the graph, of the graph database, which is most similar to the source graph.

Algorithm 2 The algorithm for matching phase

Input: The minimum DFS codes for GD: M_1; the orders of node labels and edge labels: N, E; source graph (G)
Output: Graph $G'(G' \in GD)$ which is most similar to G
1: Initialize one map structure: M_2
2: $code' \leftarrow$ get $minDFSCode(G)$ by N and E
3: **for** each record $(code, ID\_Set)$ in M_1 **do**
4: filter the minimum DFS codes of $code$ and $code'$
5: $distance \leftarrow$ calculate the $SED(code, code')$
6: $M_2.put(ID\_Set, distance)$
7: **end for**
8: Sort M_2 by distance values and return the graph IDs of the smallest record of M_2

From Algorithm 2 we can see that, there are three steps in the matching phase. The first step is to get the minimum DFS code for the source graph with the help of the orders of node labels and edge labels generated in last phase. Since this step is focus on only one graph, its computation time is very short (line 2). Secondly, we not only conduct the SED calculation of DFS codes between the source graph and the graphs of the graph database but also put the set of graph IDs and the calculated distance into a map (line 3-7). This step costs most time of the matching phase since we have to match each graph in the graph database. The time complexity of this step is $O(m * n^2)$ where m represents the number of the graphs in the graph database and n is the number of nodes the largest graph has. Generally, since m is much larger than n, the computation time of this step is durable and accepted. At last, we sort the map by the value of distance and return the graph IDs of the smallest distance (line 8).

Notice that, the SED of the minimum DFS codes between two graphs can't directly determine their similarity or distance. Suppose that the business process shown in Figure 4(a) is the source graph, Figure 4(b) and (c) show two graphs in graph database. Comparing with the source graph in (a), the graphs in (b) and (c) lack only one edge respectively, i.e., $1-2$ and $1-4$, and the rest nodes and edges are same. Thus, from the viewpoint of structure, they should have the same similarity to (a). But, according to the minimum DFS codes shown in the figure, we have $SED(a,b) != SED(a,c)$.

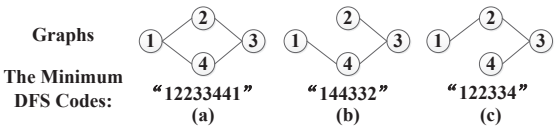


Figure 4 The illustration for filtering the minimum DFS codes

To solve this problem, we filter the minimum DFS codes (line 4) simplified in preprocessing phase. Since each edge, in a graph, represented by the DFS code has been simplified to a 2-tuple: "(l_i, l_j)", the length of a simplified minimum DFS code is the multiple of 2. Based on these tuples, we compare the codes of the source graph and the graph in database. Then, remove the same edges (DFS codes) in both of them and the rest codes are conducted SED calculation. Note that, after exchanging two nodes of an edge, if this edge is same with another edge in other graphs, these two 2-tuples could be viewed same too. Then, they would be removed from the code. As shown in Figure 4, to determine the similarity between (a) and (c), we could merely calculate the SED of two strings: "12" and "". Because, the edge "23" could be turned to "32". "34" and "41" correspond to "43" and "14" for the same rule. Based on the above discussion, from Figure 4, we can get: $SED(a,b) == SED(a,c)$.

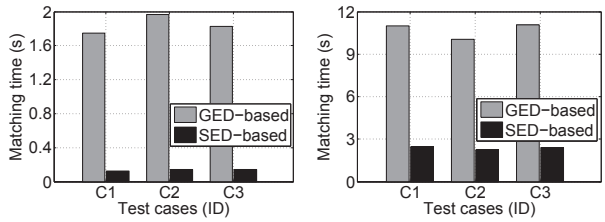
To conclude this section, we preprocess the graph database by extracting their minimum DFS codes. Then, we measure the similarity between graphs based on these DFS codes with through string edit distance technique. Furthermore, we implement a prototype based on the above details and evaluate its performance. The experimental results are present in the following section.

4. Evaluation

As mentioned before, since the online matching is much more concerned by the end users and the offline preprocessing has little contribution to the efficiency of matching, we only study the performance of the matching phase implementation in this section.

In the following experiments, we compare our method (i.e., SED based) with traditional GED-based and both of them are developed in Java (Jdk1.6). GED-based method is implemented in a fast greedy way and its time complexity is $O(n^3)$. The sorting algorithm for returning the smallest distance is bubble sort. And all the tests are done on a 2.26GHz Intel(R) Core(TM)2 Duo P8400 PC with 3GB main memory, running Windows 7. The graph dataset we used here are generated synthetically. There are totally different 26 vertex labels and each graph has a vertex size of 5 to 10. The model graph is randomly selected from this dataset. Then, we match the model graph against all data graphs in the dataset.

First of all, we study the efficiency which is measured by the time for matching. We fix the number of vertex in



(a) Size of 2000 (b) Size of 10000

Figure 5 Tests on sizes of 2000 and 10000

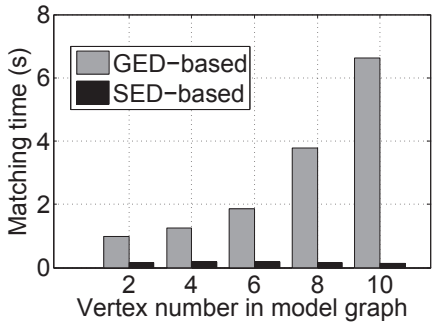


Figure 7 Matching time under different model graph sizes

the model graph to 5 and observe the matching time for both methods under different size, ranging from 2000 to 10000, of graph database. For each graph database, we use 3 different model graphs (with same vertex number 5) as different test cases. As shown in Figure 5, under different size of graph database (i.e., Figure 5(a) and Figure 5(b)), GED-based method costs much more time for matching than that of SED-based method in all test cases. This is because that the computation time for GED-based method is $O(n^3)$ while it is $O(n^2)$ for SED-based. Apparently, GED-based method would become less applicable once the size of the graph database grows large.

Table 2 shows average matching time for different graph database sizes. Clearly, as the database size increases, both methods need more time for matching. In addition, we can see from Figure 6 that the matching time of GED-based method to that of SED-based ratio

Table 2 Matching time for different graph database sizes

Database Size	GED-based (s)	SED-based (s)
2000	1.846	0.135
4000	3.380	0.432
6000	6.719	0.925
8000	8.128	1.482
10000	10.701	2.361

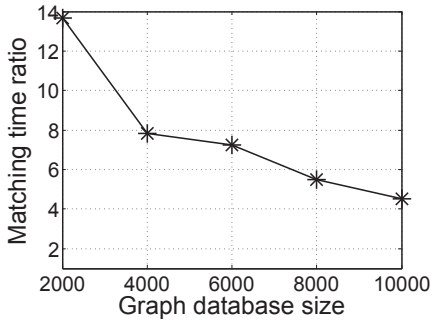


Figure 6 Matching time ratio for the model graph with 5 vertices

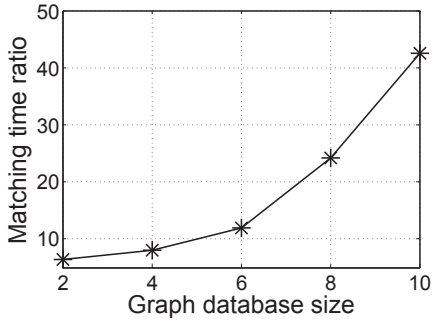


Figure 8 Matching time ratio for different model graph sizes

decreases with increasing size of graph database. Filtering the minimum DFS codes before matching is the cause for this trend.

In tests of Figure 2 and Figure 8, we fix the size of graph database to 2622 and study the efficiency under different model graphs with different vertex number ranged from 2 to 10. As shown in Figure 2, using our SED-based method, the matching time is almost unchanged with very small values of 0.1 seconds around. However, there is an apparent growth trend in the GED-based method and the matching time grows fast as the vertex number increased. That is to say, GED-based method is more sensitive to the size of the model graph than ours. The reason is that GED-based implementation has to search the best result in each step and grow based on the current best result. There are many recursive searches and judges in this procedure which costs more than computing two strings. Different from the matching ratio trend that showed in Figure 6, Figure 8 presents an opposite trend: the matching time of GED-based method to that of SED-based ratio increases with the increasing number of vertex. This is because in our SED-based method, the number of graphs which need to be matched can make more contributions to the matching time than the number of vertex of a model graph.

As for effectiveness study, based on observations on matching results of two methods, first several results for both methods are almost same. However, since our method and GED-based method adopt different principles (e.g. the costs for node/edge deletion, insertion and other operations) for similarity measurements, there exist a greater differences between matching results in a larger size for both methods. Generally, the difference ratio could be 30% around. Nevertheless, since first several results are more concerned by users, our method can satisfy their accuracy demands in general.

5. Conclusion

In this paper, we propose a novel approach for measuring similarity between graphs. Using depth-first search (DFS) strategy, we traverse the graphs and label them canonically with the minimum DFS code. Then, after extracting these codes and filtering them, we conduct the calculation of string edit distance (SED) between the source graph and graphs in database. Comparing with traditional GED based method, our approach is more efficient and the experimental evaluation proves its utility in real applications.

There is still some work needs be done in the future. For example, the accuracy of measuring similarity between graphs through calculating the SED of DFS codes should be studied both theoretically and experimentally. Besides, the relation between our proposed method and GED needs to be determined. At last, we are going to exploit our method in real graph datasets and to improve its performance making it more applicable and practicable.

Acknowledgement

This research was partially supported by following foundations: National Science and Technology Supporting Program of China(No.2012BAH06F02, No. 2011BAD21B02), National Natural Science Foundation of China under Grant (No.61272129), Research Fund for the Doctoral Program by Ministry of Education of China(No. 20110101110066)

References

[1] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J.L. Wiener, Graph structure in the Web. In Proceedings of Computer Networks. 309-320, (2000).

[2] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel. Saga: a subgraph matching tool for biological graphs. Bioinformatics, **23(2)**: 232-239, (2007).

[3] P. Willett, J. Barnard, and G. Downs. Chemical similarity searching. J. Chem. Inf. Comput. Sci, **38(6)**: 983-996, (1998).

[4] R. Dijkman, M. Dumas, and L. Garcia-Banuelos. Graph matching algorithms for business process model similarity search. In BPM, (2009).

[5] H. Bunke. On a relation between graph edit distance and maximum common subgraph. Pattern Recognition Letters, **18(8)**: 689-694, (1997).

[6] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. Pattern Recognition Letters, **19(3-4)**: 255-259, (1998).

[7] J. Raymond, E. Gardiner, and P. Willett. RASCAL: Calculation of Graph Similarity using Maximum Common Edge Subgraphs. The Computer Journal, **45(6)**: 631-644, (2002).

[8] Z. Zeng, A.K.H. Tung, J. Wang, J. Feng, and L. Zhou, Comparing Stars: On Approximating Graph Edit Distance. In Proceedings of PVLDB, 25-36, (2009).

[9] X. Yan, F. Zhu, P.S. Yu, and J. Han, Feature-based similarity search in graph structures. In Proceedings of ACM Trans. Database Syst, 1418-1453, (2006).

[10] X. Yan and J. Han, gSpan: Graph-Based Substructure Pattern Mining. In Proceedings of ICDM. 721-724. (2002).

[11] I. Levenshtein, Binary code capable of correcting deletions, insertions and reversals. Cybernetics and Control Theory, **10(8)**, 707-710, (1966).

Bin Cao is currently a Ph.D. candidate in the College of Computer Science, Zhejiang University (China). He received his B.S. from Zhejiang University of Technology, China in 2008 and he took a successive postgraduate and doctoral program in Zhejiang University since 2009. His research interests include workflow management, event processing and spatial database.

Ying Li is currently an associate professor in the College of Computer Science, Zhejiang University (China). He received his M.S. from Zhejiang University in 1997 and his Ph.D. in Computer Science from Zhejiang University in 2000. His research interests include software architecture, software automation, compiling technology and middleware technology.

Jianwei Yin is currently a professor in the College of Computer Science, Zhejiang University (China). He received his Ph.D. in Computer Science from Zhejiang University in 2001. He is the visiting scholar of Georgia Institute of Technology, America in 2008. His research interests include distributed network middleware, software architecture and information integration.