Edge Influence Computation in Dynamic Graphs

Yongrui Qin[†], Quan Z. Sheng[‡], Simon Parkinson[†], and Nickolas J.G. Falkner[$]

[†]University of Huddersfield, United Kingdom
{y.qin2,s.parkinson}@hud.ac.uk
[‡]Macquarie University, Australia
michael.sheng@mq.edu.au
[$]University of Adelaide, Australia
nickolas.falkner@adelaide.edu.au

Abstract. Reachability queries are of great importance in many research and application areas, including general graph mining, social network analysis and so on. Many approaches have been proposed to compute whether there exists one path from one node to another node in a graph. Most of these approaches focus on static graphs, however in practice dynamic graphs are more common. In this paper, we focus on handling graph reachability queries in dynamic graphs. Specifically we investigate the influence of a given edge in the graph, aiming to study the overall reachability changes in the graph brought by the possible failure/deletion of the edge. To this end, we firstly develop an efficient update algorithm for handling edge deletions. We then define the edge influence concept and put forward a novel computation algorithm to accelerate the computation of edge influence. We evaluate our approach using several real world datasets. The experimental results show that our approach outperforms traditional approaches significantly.

Keywords: Graph reachability, Dynamic graph, Edge influence

1 Introduction

Nowadays, graph structured data plays a more important role in various fields. As a foundational operation of a graph, reachability has a wide range of applications in many areas such as web data mining, biological research, social networks and computer programming, etc. It is noteworthy that, in these areas, the structure of graph is large and dynamic. To illustrate, Facebook has 1.79 billion active users monthly in the third quarter of 2016, increased from 1.55 billion active users monthly in the same period in 2015 [1]. They may have different characters such as age, gender, hobbies, and may have complicated relationship with existing users.

A large body of indexing techniques have been recently proposed to process reachability queries in graphs [2, 3, 5, 12–14]. Among them, a significant portion

[1] https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/, retrieved December 2016

of indexes are based on 2-hop labeling, which is originally proposed by Cohen et al. [7]. Most of the above mentioned approaches generally make the assumption that graphs are static. Some approaches investigate reachability in dynamic graphs, but they mainly focus on updating the overall indexes and supporting reachability queries in dynamic graphs [1, 8, 10, 15].

One important question remains open: How can we evaluate the impact of an individual edge in a large graph in terms of reachability aspect? To the best of our knowledge, there is little work available in literature about the analysis of potential impact caused by changes in a large graph like edge deletions. How to evaluate the reachability influence of an edge is still an open problem. State-of-the-art approach TOL proposed in [1], which focuses on handling reachability queries in large dynamic graphs, does not provide an efficient way to compute the reachability difference caused by the failure or deletion of an edge. Therefore it remains challenging to evaluate the impact of the deletion of a given edge efficiently.

In this work, we firstly develop a decremental maintenance algorithm to efficiently update labeling index for edge deletions. We then define edge influence to indicate the impact of an edge on the reachability of the whole graph and put forward a novel computation algorithm to calculate edge influences efficiently. Experimental results show that our method outperforms state-of-the-art approach TOL in updating indexes on edge deletions and our edge influence computation algorithm is very efficient and can scale well. Potential applications of our algorithm include finding the most influencing edges in a given network, looking to building up some most important connections in an existing network, and so on.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the details of our approach. Section 4 presents the experimental results and analysis. Section 5 concludes the paper and discusses future work.

2 Related Work

There is a large body of work on handling reachability queries in large graphs. *Tree Cover* approach is proposed by Agrawal et al. in [4], which searches a path based on a tree cover. The principle of this method is to encode each node by multiple intervals in a graph. However, even though it searches the graph efficiently, Tree Cover method can only guarantee its efficiency on static graphs. This is mainly because in order to achieve the optimal tree cover, it has to firstly establish a spanning tree when a graph changes. *Dual-Labeling* approach proposed by Wang et al. in [3] answers reachability queries by a Dual-Labeling encoding scheme. As a tree encoding method, Dual-Labeling method also encodes each node as a tree structure. Similar to Tree Cover method, it is suited for handling static graphs. *Chain Cover* approach proposed by Jagadish [6] uses a chain cover scheme to compute a reachability query. In this method, it divides a graph into several chains, which forms a chain cover for this graph.

If we want to answer a reachability query, we just search whether there exists a pair in the chain cover. The disadvantage of this method is that if the graph is dynamic, for each update operation, there are numerous pairs to be modified, which reduces the efficiency especially when the graph is large. *Path-Tree Cover* approach proposed by Jin et al. [5] answers reachability queries by using a path-tree cover scheme. Its principle is similar to the Chain Cover method and the Tree Cover method. The difference is that Path-Tree Cover uses an extra scheme to deal with non-edge in the tree structure like Dual-Labeling approach.

Meanwhile, a few approaches have been proposed for handling reachability queries in dynamic graphs [8, 10, 15]; however, these approaches cannot scale well. State-of-the-art approach in this direction is TOL, proposed by Zhu et al. [1]. TOL uses a total order labeling scheme to answer reachability queries. It encodes a level order to each node in a graph. According to this order, TOL can compute a labeling table. Comparing with other schemes mentioned above, the advantage of TOL is that it simplifies the process of table construction. TOL has an advantage in dealing with large dynamic graphs. Surprisingly, it also outperforms most existing approaches on static graphs [1]. However, the main drawback of TOL is that it can only handle node deletions but cannot handle edge deletions.

3 Methodology

In this section, we present our approach in detail. Since our approach is inspired by state-of-the-art approach TOL [1], we firstly introduce TOL index briefly, then we develop our decremental maintenance algorithm for handling edge deletions. After that, we further define the concept of edge influence to investigate the impact of an edge in the overall reachability of a graph. We also put forward an efficient computation algorithm to compute edge influence on top of the updated labeling index of the graph.

3.1 TOL Index

The TOL Index [1] of graph $\mathcal{G}$ in Fig. 1 is shown in Fig. 2. There are three columns, n (denoting nodes), $\mathcal{L}_{in}$ and $\mathcal{L}_{out}$. Note that, TOL labeling is very similar to the *2-HOP Cover* approach proposed by Cohen et al. [7].

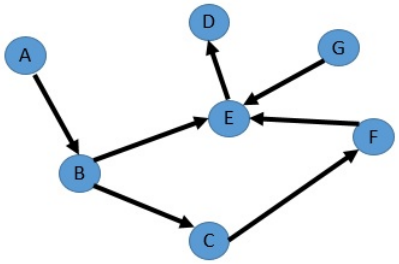


Fig. 1. Graph $\mathcal{G}$ (DAG)

n	L_{out}	L_{in}
A	{A, C}	{A}
B	{B,C}	{A, B}
C	{C}	{C}
D	{D}	{C,D,E}
E	{E}	{C,E}
F	{E,F}	{C,F}
G	{E,G}	{G}

Fig. 2. TOL Index $\mathcal{L}$

Example 1. According to Fig. 2, for node C, Column $\mathcal{L}_{out}$ contains {C}. It means that node C can reach all nodes that contains C in Column $\mathcal{L}_{in}$, including nodes C, D, E, and F. Similarly, for node E, Column $\mathcal{L}_{in}$ contains $\{C, E\}$. Hence node E can be reached by all nodes in Column $\mathcal{L}_{out}$ that contains C or E, including nodes A, B, C, E, and F.

3.2 Handling Edge Deletions

Suppose the TOL index of a given graph $\mathcal{G}$ is $\mathcal{L}$. Assuming after deleting edge e from $\mathcal{G}$, we have $\mathcal{G}'$ and its corresponding index $\mathcal{L}'$. Using TOL approach, we have to compute $\mathcal{L}'$ from scratch for $\mathcal{G}'$. It is obvious that such process is not efficient. To improve efficiency, we devise a new approach that only calculates the difference between labeling indexes $\mathcal{L}$ and $\mathcal{L}'$, if an edge deletion occurs.

Given a DAG $\mathcal{G} = (V, E)$, if deleting edge $e = N_S \rightarrow N_E \in E$ from it, we can construct a new graph $\mathcal{G}_r \in \mathcal{G}$, which includes only nodes that can reach or be reached by either node N_S or node N_E. The structure of $\mathcal{G}_r$ is presented in Fig. 3.

It should be noted that, we only need to deal with those nodes whose reachability status to another node might be changed due to the edge deletion. All such nodes can be divided into six sets, N_{toSO}, N_{toEO}, N_{fromSO}, N_{fromEO}, N_{toSE} and N_{fromSE}. Details of these node sets are as follows:

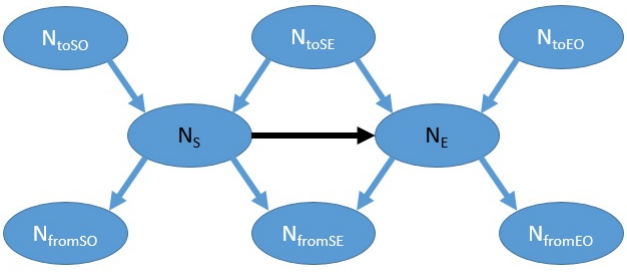


Fig. 3. Graph $\mathcal{G}_r$

N_{toSO} contains nodes that can reach node N_S but cannot reach node N_E, defined as a special ancestor node set of N_S.
N_{toEO} contains nodes that can reach node N_E but cannot reach node N_S, defined as a special ancestor node set of N_E.
N_{fromSO} contains nodes that can be reached by node N_S but cannot be reached by node N_E, defined as a special descendant node set of N_S.
N_{fromEO} contains nodes that can be reached by node N_E but cannot be reached by node N_S, defined as a special descendant node set of N_E.
N_{toSE} contains nodes that can reach both node N_S and node N_E, which form the common ancestor node set of both N_S and N_E.
N_{fromSE} contains nodes that can be reached by both node N_S and node N_E, which form the common descendant node set of both N_S and N_E.

The next process is to classify all relevant nodes into the corresponding sets, which is described in the following.

1. Find all ancestor and descendant node sets of N_S and N_E using BFS (Breadth-First-Search).
2. Compare the ancestor node set of N_S with the ancestor node set of N_E; put all nodes that appear in both ancestor node sets together and form N_{toSE}. Then N_{toSO} is the set containing the rest ancestor nodes of N_S and N_{toEO} is the set containing the rest ancestor nodes of N_E.
3. Following a similar step to the above Step 2, we can construct all the defined descendant node sets, including N_{fromSE}, N_{fromSO} and N_{fromEO}.

After completing the nodes classification, we can start to handle the edge deletion process. When deleting edge $e = N_S \rightarrow N_E \in E$, changes of each node in the labeling index can be divided into three situations as shown in Fig. 4.

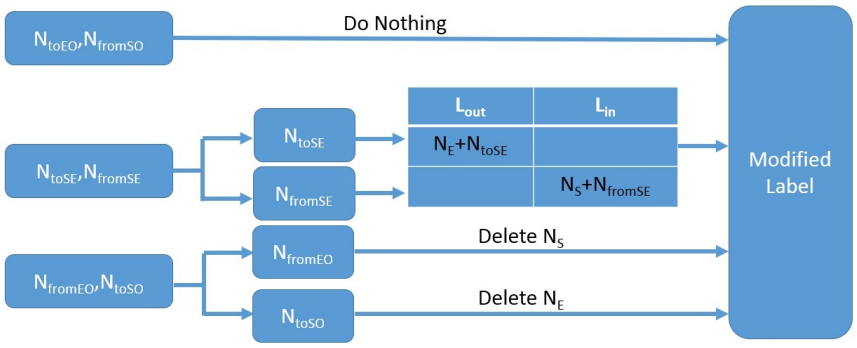


Fig. 4. Edge Deletion Flow Chart

Situation 1: For node sets N_{toEO} and N_{fromSO}, no changes are needed in the original labeling index. As shown in Fig. 3, a path from a node of N_{toEO} to any other node of $\mathcal{G}_r$ or a path from any node of $\mathcal{G}_r$ to a node in N_{fromSO}, will not contain edge e. In other words, reachability of any node in N_{toEO} and N_{fromSO} will not be affected by deletion of edge e.

Situation 2: For node sets N_{toSE} and N_{fromSE}, although the reachability related to the nodes in these sets is the same as before, the path of reachability may be different. This creates impact on the labeling index. We use the following steps to modify the labeling index to ensure the reachability of these nodes remains the same as before when computed from the updated labeling index.

1. Denote node set of $\mathcal{L}_{out}$ column of node N_E in the labeling index as N_{Eout}.
2. Add N_{Eout} to $\mathcal{L}_{out}$ column of each node in N_{toSE} in the labeling index.
3. Denote node set of $\mathcal{L}_{in}$ column of N_S in the labeling index as N_{Sin}.
4. Add N_{Sin} to $\mathcal{L}_{in}$ column of each node in N_{fromSE} in the labeling index.

For the three node sets that N_{toSE} can reach (including N_{fromSO}, N_{fromSE} and N_{fromEO}), the reachability status from set N_{toSE} to set N_{fromSO} is not affected by edge e (see Fig. 3). For N_{fromSE} and N_{fromEO}, they are the only sets that can be reached by node N_E. Therefore, we only need to add N_{Eout} to the column of each node of N_{fromSE} and N_{fromEO} in the new labeling index.

Situation 3: For node sets N_{fromEO} and N_{toSO}, whether edge e is deleted or not has significant influence on their reachability status. After deleting edge e, all nodes in N_{toSO} cannot reach node N_E any more. When deleting edge e, the update of the labeling index can be achieved by deleting node N_E in the column of $\mathcal{L}_{in}$ of each node in N_{toSO}. It is similar to N_{toSO}, for N_{fromEO}, we only need to delete node N_S in the column of $\mathcal{L}_{out}$ of each node of N_{fromEO}.

Similar to deleting a node in TOL approach [1], the key step of updating index in the above situation is to modify the labeling index as follows.

For N_{toSO} and N_S:

1. Check the column $\mathcal{L}_{out}$ of node N_S, if there is node N_E, delete it.
2. Find out all $\mathcal{L}_{out}$ of child nodes of N_S except N_E, and denote them as set Set_S. Add all nodes in Set_S into $\mathcal{L}_{out}$ of N_S except the case that this node is already in $\mathcal{L}_{out}$.
3. Find out all nodes in N_{toSO}, and apply steps similar to Steps 1 to 2.

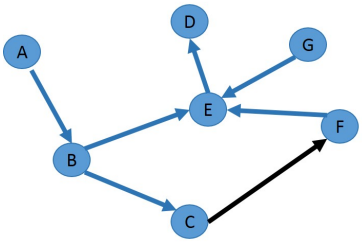
For N_{toEO} and N_E:

1. Check the column $\mathcal{L}_{in}$ of node N_E, if there is node N_S, delete it.
2. Find out all $\mathcal{L}_{in}$ of father nodes of N_E except N_S, and denote them as set Set_E. Add all nodes in Set_E into $\mathcal{L}_{in}$ of N_E except the case that this node is already in $\mathcal{L}_{in}$.
3. Find out all nodes in N_{toEO}, and apply steps similar to Steps 1 to 2.

Example 2. Given a DAG $\mathcal{G}$ and its corresponding labeling index $\mathcal{L}$ as shown in Fig. 4, if deleting edge $e = C \rightarrow F$, the process of edge deletion is as follows.

Firstly, we divide all nodes of $\mathcal{G}$ into six sets, where:

N_{toSO} contains nodes A and B.
N_{fromEO} contains nodes E and D.
N_{toEO}, N_{toSE}, N_{fromSO}, N_{fromSE} All are empty sets.



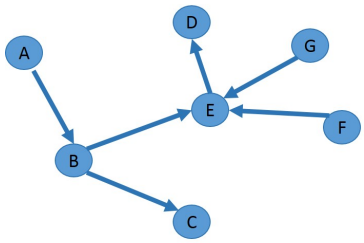
n	L_{out}	L_{in}
A	{A, C}	{A}
B	{B,C}	{A, B}
C	{C}	{C}
D	{D}	{C,D,E}
E	{E}	{C,E}
F	{E,F}	{C,F}
G	{E,G}	{G}

Fig. 5. Graph $\mathcal{G}$ and labeling index $\mathcal{L}$

It is obvious that we need to handle N_{toSO} and N_{fromEO}, which is Situation 3.

1. After checking column $\mathcal{L}_{out}$ of node C, we find that there is no F, so we do not need to delete anything.
2. We need to find out the child node of C, which is node F. Because F is the terminus node of edge e, we do not need to add anything to $\mathcal{L}_{out}$.
3. We deal with the father node of C, which is B. Because F is not in $\mathcal{L}_{out}$ of B, we just need to check all child nodes of B, E and C. For child node C, the $\mathcal{L}_{out}$ of C has already been included in $\mathcal{L}_{out}$ of B, which we do not need deal with. For child node E, the $\mathcal{L}_{out}$ of E is not in $\mathcal{L}_{out}$ of B, so we need to add E to $\mathcal{L}_{out}$ of B.
   The new $\mathcal{L}'_{out}$ of B = $\mathcal{L}_{out}$ of B + $\mathcal{L}_{out}$ of E = $\{B, E, C\}$.
4. We handle with the father node of B, which is A. Similar to B, we can compute the new $\mathcal{L}'_{out}$ of $A = \{A, B, C, E\}$. Because A does not have any father node, the process of dealing with C finishes.
5. Similar to C, after handling F, we can compute the new $\mathcal{L}'_{in}$ of $F = \{F\}$.
6. Similar to nodes B and A, we have $\mathcal{L}'_{in}$ of $E = \{E\}$ and $\mathcal{L}'_{in}$ of $D = \{D, E\}$. Because D does not have any child node, the process of dealing with F finishes.

Finally we can successfully compute the new $\mathcal{L}'$ of $\mathcal{G}'$ as shown in Fig. 6.



n	L_{out}	L_{in}
A	{A, B, E, C}	{A}
B	{B, E, C}	{A, B}
C	{C}	{C}
D	{D}	{D,E}
E	{E}	{E}
F	{E,F}	{F}
G	{E,G}	{G}

Fig. 6. New graph $\mathcal{G}'$ and new labeling index $\mathcal{L}$

3.3 Edge Influence

Next, we define edge influence and show how to calculate influence of a given edge efficiently. The influence provides a measure of how important an edge is

to a graph. In other words, without this edge, its influence shows how greatly the reachability of all pairs of nodes of this graph will change.

Definition 1. *When one edge is deleted, the number of pairs of nodes whose reachability has been changed due to the deletion stands for the absolute influence of this edge (denoted as $\overline{Inf_e}$).*

According to Definition 1, we can calculate the maximum absolute influence of an edge in a given graph according to the following theorem.

Theorem 1. *Whatever the structure of the graph is, provided the amount of nodes is n, the maximum absolute influence (denoted as $\overline{Inf_{max}}$) of any edge in the graph is:*

$$\overline{Inf_{max}} = \begin{cases} \left(\frac{n}{2}\right)^2 & n \text{ is even.} \\ \frac{(n-1)}{2} \cdot \frac{(n+1)}{2} & n \text{ is odd.} \end{cases} \tag{1}$$

Proof. Given a graph $\mathcal{G} = (V, E)$ with n nodes, there must exist an edge $e \in E$ which divides V into three node sets A, B and C. Sets A and B meet two requirements: First, each node in A can find a path to each node in B; Second, every such path contains e. Set C contains all other nodes that do not belong to sets A and B.

If denoting the number of nodes in sets A, B and C as a, b and c, we have formula (2).

$$n = a + b + c \tag{2}$$

If deleting edge e, any path from one node in A to another node in B may become disconnected. And the amount of these paths are $a \cdot b$. Then according to Definition 1, we have formula (3).

$$\overline{Inf_e} = ab \tag{3}$$

Combining (2) and (3), we have formula (4) as follows.

$$\overline{Inf_e} = a(n - a - c) \tag{4}$$

It is obvious that when c increases, the value of Inf_e decreases. In order to achieve the maximum value of Inf_e, c should be 0. Then, we can transform (4) to (5).

$$\overline{Inf_e} = a(n - a) \tag{5}$$

Obviously, Inf_e is maximum when $a = \frac{n}{2}$. Since $n \in \mathbb{Z}^+$, if n is even, $a = \frac{n}{2}$, and if n is odd, $a = \frac{n-1}{2}$. Combining (2), we arrive at formula (1). This completes the proof. □

Once $\overline{Inf_{max}}$ of a graph is known, the normalized influence of an edge in this graph can be calculated as follows.

Definition 2. *Given a graph $\mathcal{G}' = (V, E)$ with n nodes, the influence of an edge $e \in E$ is denoted as Inf_e, which can be computed as follows:*

$$Inf_e = \frac{\overline{Inf_e}}{\overline{Inf_{max}}} \tag{6}$$

Calculation Given a DAG $\mathcal{G}$, if deleting an edge $e = N_S \rightarrow N_E$, we can apply the following steps to calculate the influence of edge e.

- According to Theorem 1, we calculate $\overline{Inf_{max}}$
- Find out the node set where nodes can be reached by node N_E including node N_E itself, denoted as Set_E
- Find out nodes that belong to Set_E and can be reached by node N_S before deleting edge e, denoted as Set_{SP}
- Find out nodes that belong to Set_E and can be reached by node N_S after deleting edge e, denoted as Set_{SL}. And then we can get node set $Set_S = Set_{SP} - Set_{SL}$. The amount of nodes in Set_S equals to the amount of bode pairs with changed reachability status that is related to node N_S after deleting edge e
- For each ancestor node of N_S, we can calculate the corresponding Set_i. It should be noted that during the calculation, once we have $Set_i = \emptyset$, we can stop calculating ancestors of node N_i
- Assuming n_i is the amount of nodes in Set_i, the absolute influence of edge e is $\overline{Inf_e} = \sum n_i$. Then we can compute the influence of edge e according to formula (6)

Example 3. Given a DAG $\mathcal{G}$, we can calculate the influence of edge $e = C \rightarrow F$ in the following.

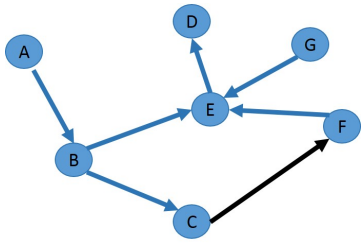


Fig. 7. Graph $\mathcal{G}$

- Calculate the maximum absolute influence $\overline{Inf_{max}}$ of graph $\mathcal{G}$.

$$\overline{Inf_{max}} = \frac{(7-1)}{2} \times \frac{(7+1)}{2} = 12$$

- There are two other nodes that F can reach. Then $Set_F = \{D, E, F\}$

- $Set_{CP} = \{D, E, F\}$
- $Set_{CL} = \varnothing$ Then $Set_C = \{D, E, F\} - \varnothing = \{D, E, F\}$. Set_C has three nodes D, E and F, so $n_C = 3$
- For node C's ancestor node B, $Set_B = \{D, E, F\} - \{D, E\} = \{F\}$. There is only one node F in Set_B, so $n_B = 1$. For the second ancestor node A of node C, $\underline{Set_A} = \{F\} - \varnothing = \{F\}$, which means $n_A = 1$
- $Inf_e = \sum n_i = n_C + n_B + n_A = 3 + 1 + 1 = 5$. Then

$$Inf_e = \frac{\overline{Inf_e}}{\overline{Inf_{max}}} = \frac{5}{12}$$

Therefore, the influence of edge e is $\frac{5}{12}$ in graph $\mathcal{G}$ in Fig. 7.

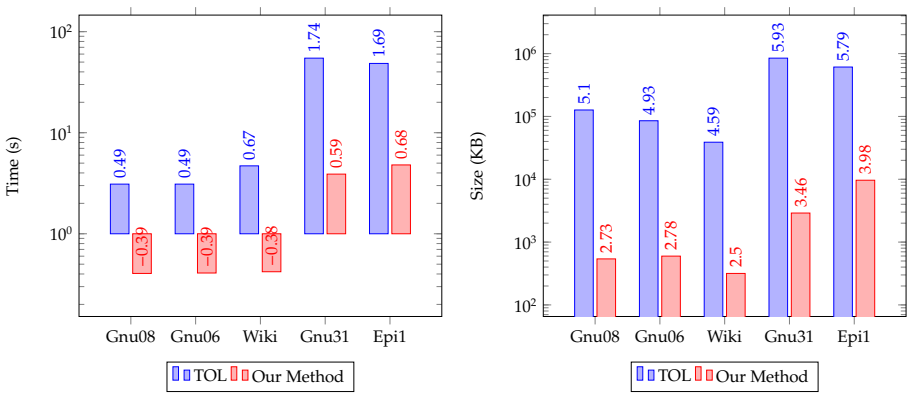
4 Experiments

We use the following five real-world datasets in our experiments: p2p-Gnutella08 (*Gnu08*, 6.3K nodes, 21K edges), p2p-Gnutella06 (*Gnu06*, 8.7K nodes, 32K edges), Wiki-Vote (*Wiki*, 7.1K nodes, 104K edges), p2p-Gnutella31 (*Gnu31*, 63K nodes, 148K edges) and soc-Epinions1 (*Epi1*, 76K nodes, 509K edges) [11]. We use these datasets to conduct two sets of experiments. The first set is to delete 100 edges generated by the graph transformation module randomly. We compare labeling index method (TOL method) with our method by performing 100 edge deletions and record the average time cost and the average index size (the changed part). The second set is to validate our edge influence algorithm also by performing 100 edge deletions and record the average time cost. All experiments were performed on a PC with 64-bit Windows 7, 8GB RAM and 2.40GHZ Intel i7-3630QM CPU.

It should be noted that these bar charts use log-scale plotting features. From Fig. 8(a), we can see that our method performs deletion nearly at a speed of an order of magnitude faster than TOL method among all the datasets. The updated index size of our method is also much smaller than TOL method (see Fig. 8(b)). The main reason for these two experimental results is that our method can incrementally compute updated index for edge deletions, while TOL method can only support node deletions. For edge deletions, TOL method has to recompute the whole index from scratch. Here, we do not compare indexing time with other approaches for static graphs, as TOL method has been shown to outperform most existing approaches for static graphs [1].

Fig. 9 shows the average calculation time of edge influences for the 100 deleted edges. Since this is the first attempt on the calculation of edge influence, we compare our method with a modified BFS&DFS method[2]. From the figure we can see that our method can compute edge influences orders of magnitude faster than the modified BFS&DFS method. For dataset *Epi1*, the modified BFS&DFS method cannot complete the calculation within 24 hours.

[2] BFS&DFS refers to Breadth-First-Search and Depth-First-Search. Both our method and BFS&DFS were performed on top of the updated labeling index.



(a) Time for Updating Index (b) Size of Updated Part of the Index

Fig. 8. Comparison with TOL Approach

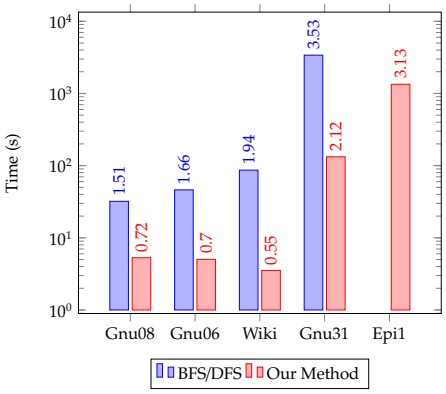


Fig. 9. Calculation Time of Edge Influence

5 Conclusion

In this work, we propose a new approach to calculate an updated labeling index of a graph after edge deletions. Then we define the influence of edges in a graph based on reachability between nodes affected by the potential deletions of edges, and provide an approach to calculating the influence of any given edge in the graph. Our experiments validate the efficiency and scalability of our approach.

Future work includes devising faster algorithms to handle even larger dynamic graphs (e.g., at the scale of millions of nodes and edges) and design new algorithms to rank all the edges based on our defined edge influence in large graphs.

Acknowledgments

Authors would like to thank Xiaorong Liang for the implementation of the algorithms and thank anonymous reviewers for their valuable comments.

References

[1] Zhu, Andy Diwen, et al. "Reachability Queries on Large Dynamic Graphs: A Total Order Approach."

[2] Cheng, James, et al. "TF-Label: a topological-folding labeling scheme for reachability querying in a large graph." Proceedings of the 2013 international conference on Management of data. ACM, 2013.

[3] Wang, Haixun, et al. "Dual labeling: Answering graph reachability queries in constant time." Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on. IEEE, 2006.

[4] Agrawal, Rakesh, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. Efficient management of transitive relationships in large data and knowledge bases. Vol. 18. No. 2. ACM, 1989.

[5] Jin, Ruoming, et al. "Efficiently answering reachability queries on very large directed graphs." Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008.

[6] Jagadish, H. V. "A compression technique to materialize transitive closure." ACM Transactions on Database Systems (TODS) 15.4 (1990): 558-598.

[7] Cohen, Edith, et al. "Reachability and distance queries via 2-hop labels." SIAM Journal on Computing 32.5 (2003): 1338-1355.

[8] Bramandia, Ramadhana, Byron Choi, and Wee Keong Ng. "Incremental maintenance of 2-hop labeling of large graphs." Knowledge and Data Engineering, IEEE Transactions on 22.5 (2010): 682-698.

[9] Schenkel, Ralf, Anja Theobald, and Gerhard Weikum. "HOPI: An efficient connection index for complex XML document collections." Advances in Database Technology-EDBT 2004. Springer Berlin Heidelberg, 2004. 237-255.

[10] Yildirim, Hilmi, Vineet Chaoji, and Mohammed J. Zaki. "Dagger: A scalable index for reachability queries in large dynamic graphs." arXiv preprint arXiv:1301.0977 (2013).

[11] Leskovec, Jure. "Stanford large network dataset collection, 2014." URL: http://snap.stanford.edu/data/index.html

[12] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. PVLDB, 3(1):276284, 2010.

[13] S. J. van Schaik and O. de Moor. A memory efcient reachability data structure through bit vector compression. In SIGMOD, pages 913924, 2011.

[14] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. Ferrari: Flexible and efcient reachability range assignment for graph indexing. In ICDE, pages 10091020, 2013.

[15] R. Schenkel, A. Theobald, and G. Weikum. Efcient creation and incremental maintenance of the hopi index for complex xml document collections. In ICDE, pages 360371, 2005.