# TopCom: Index for Shortest Distance Query in Directed Graph

VACHIK S. DAVE, Indiana University Purdue University, Indianapolis
MOHAMMAD AL HASAN, Indiana University Purdue University, Indianapolis



Finding shortest distance between two vertices in a graph is an important problem due to its numerous applications in diverse domains, including geo-spatial databases, social network analysis, and information retrieval. Classical algorithms (such as, Dijkstra) solve this problem in polynomial time, but these algorithms cannot provide real-time response for a large number of bursty queries on a large graph. So, indexing based solutions that pre-process the graph for efficiently answering (exactly or approximately) a large number of distance queries in real-time is becoming increasingly popular. Existing solutions have varying performance in terms of index size, index building time, query time, and accuracy. In this work, we propose TOPCOM, a novel indexing-based solution for exactly answering distance queries. Our experiments with two of the existing state-of-the-art methods (IS-Label and TreeMap) show the superiority of TOPCOM over these two methods considering scalability and query time. Besides, indexing of TOPCOM exploits the DAG (directed acyclic graph) structure in the graph, which makes it significantly faster than the existing methods if the SCCs (strongly connected component) of the input graph are relatively small.

CCS Concepts: •**Information systems** → **Information retrieval query processing;** *Data structures;* Information storage systems;

General Terms: Graph Algorithms, Performance

Additional Key Words and Phrases: Shortest Distance Query, Indexing method for Distance Query, Directed Acyclic Graph

## 1. INTRODUCTION

Finding shortest distance between two nodes in a graph (*distance query*) is one of the most useful operations in graph analysis. Besides the application that stands for its literal meaning, i.e. finding the shortest distance between two places in a road network, this operation is useful in many other applications in social and information networks. For instance, in social networks, the shortest path distance is used in the calculation of different centrality metrics, including closeness centrality and betweenness centrality [Okamoto et al. 2008; Erdem Sariyuce et al. 2013]. It is also used as a criterion for finding highly influential nodes [Kempe et al. 2003], and for detecting communities in a network [Backstrom et al. 2006]. Scientists have also used shortest path distance to generate features for predicting future links in a network [Hasan and Zaki 2011]. In information networks, shortest path distance is used for keyword search [Kargar and An 2011], and also for relevance ranking [Ukkonen et al. 2008].

Due to the importance of the shortest path distance problem, researchers have been studying this problem from the ancient time, and several classical algorithms (Dijkstra, Bellman-Ford, Floyd-Warshall) exist for this problem, which run in polynomial time over the number of vertices and the number of edges of the network. However, as real-life graphs grow in the order of thousands or millions of vertices, classical algorithms deem inefficient for providing real-time answers for a large number of distance queries on such graphs. For example, for a graph of a few thousand vertices, a contemporary desktop computer takes an order of seconds to answer a single query, so thousands of queries take tens of minutes, which is not acceptable for many real-time applications. So, there is a growing interest for the discovery of more efficient methods for solving this task.

Author's addresses: V. S. Dave, Computer & Information Science Department, Indian University Purdue University, Indianapolis; M. Al Hasan, Computer & Information Science Department, Indian University Purdue University, Indianapolis.



This is the author's manuscript of the article published in final edited form as:
Dave, V. S., & Hasan, M. A. (2015). TopCom: Index for Shortest Distance Query in Directed Graph. In Database and Expert Systems Applications (pp. 471–480). Springer, Cham. https://doi.org/10.1007/978-3-319-22849-5_32

Various approaches are considered for obtaining an efficient distance query method for large graphs. One of them is to exploit topological properties of real-life networks that adhere to some specific characteristics. For instance, many researchers exploit the spatial and planar properties of road networks [Tao et al. 2011; Abraham et al. 2011; Yan et al. 2013] to obtain efficient solutions for distance queries in road networks. However, for a general network from any other domain, such methods perform poorly [Abraham et al. 2012]. The second approach is to perform pre-processing on the host graph and build an index data structure which can be used at runtime to answer the distance query between an arbitrary pair of nodes more efficiently. Several indexing ideas are used, but two are the most common, landmark-based indexing [Tretyakov et al. 2011; Qiao et al. 2014; Potamias et al. 2009; Akiba et al. 2013] and 2-hop-cover indexing [Cohen et al. 2002]. Methods adopting the former idea identify a set of landmark nodes and pre-compute all-single source shortest paths from these landmark nodes. During query time, distances between a pair of arbitrary nodes are answered from their distances to their respective closest landmark nodes. Most of these methods deliver an approximation of shortest path distance except a method presented in [Akiba et al. 2013]. Methods adopting the two-hop cover indexing generally find the exact solution for a distance query [Jin et al. 2012b; Jiang et al. 2014; Fu et al. 2013]. These methods store a collection of hops (paths starting from that node), such that the shortest path between a pair of arbitrary vertices can be obtained from the intersection of the hops of those vertices.

A related work to the shortest path problem is the reachability problem. Given a directed graph $G(V, E)$, and a pair of vertices $u$ and $v$, the reachability problem answers whether a path exists from $u$ to $v$. This problem can be solved in $O(|V|+|E|)$ time using graph traversal, where $V$ is the set of vertices and $E$ is the set of edges. However, using a reachability index, a better runtime can be obtained in practice. All the existing solutions [Yildirim et al. 2012; Zhu et al. 2014] of the reachability problem solve it for a directed acyclic graph (DAG). This is due to the fact that any directed graph can be converted to a DAG such that a DAG node is a strongly connected component (SCC) of the original graph; since any nodes in an SCC is reachable to each other, the reachability solution in the DAG easily answers a reachability query in the original graph. The indexing idea that we propose in this work also exploits the SCC, but unlike existing works we solve the distance query problem instead of reachability.

In this work, we propose TOPCOM [1], an indexing based method for obtaining exact solution of a distance query in an arbitrary directed graph. In principle, TOPCOM uses a 2-hop-cover solution, but its indexing is different from other existing indexing methods. Specifically, the basic indexing scheme of TOPCOM is designed for a DAG and it is inspired from the indexing solution of the reachability queries proposed in [Cheng et al. 2013]. Due to its design, TOPCOM exhibits a very attractive performance for a DAG or general graph in which SCCs are relatively small. However, we also extend the basic indexing scheme so that it also solves the distance query for an arbitrary directed graph. We show experiment results that validate TOPCOM's superior performance over IS-Label [Fu et al. 2013] and TreeMap [Xiang 2014] which are two of the fastest known indexing based shortest path methods in the recent years. Following other recent works, we also compare our method with bi-directional Dijkstra, which is a well-accepted baseline method for distance query solutions in a directed graph. Note that, this journal article is an extended version of a published conference article [Dave and Hasan 2015]; the conference article works for DAG only, but this work solves distance query indexing for arbitrary directed graphs.

[1] TOPCOM stands for **Top**ological **Com**pression which is the fundamental operation that is used to create the index data structure of this method.

## 2. RELATED WORKS

Shortest distance on a graph has many interesting recent and earlier works. In the section we discuss the most important works among these under two categories: (1) Online shortest distance calculation, (2) Offline (Index based) shortest distance calculation.

### 2.1. Online shortest distance calculation:

For unweighted graph, the simplest online method to find shortest distance is Breadth First Search (BFS) with time complexity $O(|V| + |E|)$, where $V$ is number of vertices and $E$ is number of edges of the graph. For weighted graph, most well-known single source shortest distance algorithm is Dijkstra's algorithm, which computes shortest distance for weighted graph with positive weights. Using a binary heap based priority queue, the complexity of Dijkstra's algorithm is $O(|E| \lg |V|)$ and the same using a Fibonacci heap is $O(|E| + |V| \lg |V|)$. Another well known algorithm for single source shortest path is the Bellman-Ford algorithm [Bellman 1958; Cormen et al. 2001] with time complexity $O(|V| \cdot |E|)$, which is generally slow for large graphs with millions of nodes and edges.

There are methods proposed by different researchers to improve the above classical shortest distance methods [Bauer et al. 2010; Wagner and Willhalm 2007]. Although, they do not improve the worst case complexity of the shortest path algorithm, they do exhibit good average-case behavior. The most popular among these methods is Bidirectional Dijkstra [Sint and de Champeaux 1977], which is particularly applicable when the objective is to obtain the shortest distance between a pair of vertices. The computational complexity of bidirectional search can be denoted as $O(b^{d/2})$, where $b$ is the branching factor and $d$ is the distance from start node to target node. Real life networks have small value of $d$ (typically smaller than 10)—a fact that makes this algorithm an attractive choice for many applications. In this work, bidirectional Dijkstra is one of the methods with which we compare our proposed solution.

### 2.2. Offline (Index based) shortest distance calculation:

For large graphs, online methods are slower than an indexing based method, so most of the recent research efforts are concentrated towards indexing based methods. The literature for shortest distance indexing is quite vast, so, we review few of the works that have published in the recent years. For a detailed review, we refer the readers to read [Zwick 2001; Sommer 2014].

Many of the existing works for shortest distance computation is specifically designed for the **road networks** [Yan et al. 2013; Rice and Tsotras 2010; Tao et al. 2011; Zhu et al. 2013a; Abraham et al. 2011; Geisberger et al. 2008; Sanders and Schultes 2005; Jung and Pramanik 2002]. Such networks show hierarchical structures with the presence of junctions, hubs, and highways; the shortest distance computation methods for these networks exploit the hierarchical structure for compressing distance matrix or for building distance indices [Rice and Tsotras 2010; Geisberger et al. 2008]. For example, Sanders et al. [Sanders and Schultes 2005] use highway hierarchy and design an exact shortest distance computation method that runs faster than a method that does not use the hierarchy structure. Zhu et al. [Zhu et al. 2013a] design a hierarchy based indexing and prove that the results on real-life graphs is close to the theoretical complexity of the proposed method. Jung et al. [Jung and Pramanik 2002] design an efficient shortest path computation method for hierarchically structured topographical road maps. Abraham et al. [Abraham et al. 2011] have proposed an efficient hub-based labeling (HL) method to answer shortest path distance query on road networks. Tao et al. [Tao et al. 2011]

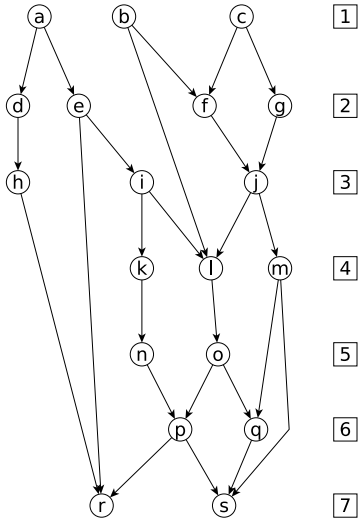
explore the spatial property and find k-skip graph which can answer k-skip shortest path i.e. path created from the original shortest path by skipping at-most k consecutive nodes. Recently Yan et al. [Yan et al. 2013] propose a method to find the distance preserving sub-graphs to answer a shortest distance query more efficiently. However, most of the indexing schemes for the road networks are based on some specific property of the road networks and they are ineffective for general graphs that do not satisfy those properties of road networks [Abraham et al. 2012].

Finding exact shortest distance in a large graph is a costly task, hence few researchers have proposed methods for computing **estimated shortest distance** [Tretyakov et al. 2011; Gubichev et al. 2010; Qiao et al. 2014; Potamias et al. 2009]. The most common among the estimated shortest distance based methods is the landmark based method, which selects a set of landmark nodes based on some criteria and finds shortest paths that must go through those landmark nodes. The main task here is to decide the set of vertices that are optimal choice as landmarks. However, it has been shown that this optimization problem is $\mathcal{NP}$-Hard [Potamias et al. 2009], so researchers adopt various heuristics based approaches for choosing those landmarks. Potamias et al. [Potamias et al. 2009] compare centrality and degree based approaches for selecting landmarks. Gubichev et al. [Gubichev et al. 2010] propose a sketch based indexing method for estimating answer of a shortest distance query. Treyakov et al. [Tretyakov et al. 2011] propose a landmark based fully dynamic approximation method using shortest path tree and also obtain an improved set of vertices as landmark; they show that their method has less approximation error than other landmark based approaches. Qiao et al. [Qiao et al. 2014] propose a query based local landmark method which selects landmark nodes that are local to the query in the sense that the obtained shortest path is the closest to the real shortest path as much as possible; this method also improves the estimation accuracy. Our proposed indexing method, TOPCOM, is not comparable to these methods, because unlike these methods, our method provides exact shortest path distance.

There are some other works for finding shortest distance in large graphs which are proposed very recently; examples include [Fu et al. 2013; Zhu et al. 2013b; Jin et al. 2012b; Cheng and Yu 2009; Abraham et al. 2012; Xiang 2014; Akiba et al. 2013; Gao et al. 2011; Wei 2010; Cheng et al. 2012]. Many of these have unique ideas, so it is difficult to categorize them under a generic shortest path method. For example, Gao et al. [Gao et al. 2011] use a relational approach and propose an index called SegTable which stores local segments of a shortest distance. Zhu at al. [Zhu et al. 2013b] propose a method to answer single source shortest distance query for a huge graph on disk. Akiba et al. [Akiba et al. 2013] [2] propose a unique pruning method based on degree of a vertex, which can efficiently reduce the search space of BFS. Highway centric label (HCL) [Jin et al. 2012b] is one of the fastest recent methods that is proposed for a shortest distance query on both directed and undirected graphs. In a follow-up work, Xiang proposes TreeMap [Xiang 2014], a tree decomposition based approach for solving distance query exactly; the author compares TreeMap's solution with those of HCL to show that the former has better performance. Another recent method is called IS-Label which is proposed by Fu et al. [Fu et al. 2013]. They have also shown that that IS-Label has superior performance than HCL. In this work, we compare TOPCOM with both IS-Label and TreeMap, which are among the best of the existing index based methods.

Our proposed method exploits conversion of a directed graph into a *directed acyclic graph (DAG)* by collapsing *strongly connected components* (SCCs) into a ver-

[2]This work cannot be compared with TOPCOM, because the authors were unable to provide code that can answer shortest distance query in directed graphs.




| Edge | *distance* |
|---|---|
| $(d, h')$ | 2 |
| $(e, k)$ | 2 |
| $(e, l)$ | 2 |
| $(f, l)$ | 2 |
| $(f, m)$ | 2 |
| $(g, l)$ | 2 |
| $(g, m)$ | 2 |
| $(k, p)$ | 2 |
| $(l, p)$ | 2 |
| $(l, q)$ | 2 |

Fig. 1: Pre-processing of DAG before Compression: (a) Original DAG $G$ and (b) Modified DAG $G_m$. The dummy edges data structure ($DummyEdges$) associated with this modified DAG is shown to the right.

tex. It is a widely used approach for solving **reachability query** task in a directed graph [Cheng et al. 2013; Jin et al. 2009; Yildirim et al. 2012; Jin et al. 2008; Jin et al. 2012a; Cheng and Yu 2009]. Any two nodes in an SCC are reachable from each other, hence for a directed graph the reachability query between two nodes can be answered through the reachability answer between their corresponding DAG nodes. However, note that, a reachability query is easier than a distance query, because the latter provides distance value as answer, which is relatively harder as the graph can be weighted. Specifically, for online (non-indexing) solution, reachability has a complexity of $O(|V|+|E|)$, and shortest distance query for weighted graph has a higher complexity, $O(|E| \lg |V|)$. Authors in [Cheng et al. 2013] uses topological folding of DAG for answering reachability query. Our proposed method TOPCOM also uses topological folding property of DAG to compress the DAG level-wise, but unlike the above work, our work answers distance queries on weighted graph. Cheng et al. [Cheng and Yu 2009] is another existing work which also proposes a DAG based approach for answering distance queries by finding distance aware 2-HOP cover.

Note that, one of the known limitations of indexing based methods is that they require more memory, but this is not a concern for TOPCOM with today's commodity machine having main memory in multiples of 2 GB.

## 3. METHOD

In this section, we discuss the shortest distance indexing of TOPCOM for a DAG. In subsequent section, we will show how this can be adapted for a general directed graph.

### 3.1. Topological compression

The main idea of TOPCOM is based on topological compression of DAG, which is performed during the index building step. During the compression, additional distance information is preserved in a data structure which TOPCOM uses for answering a dis-

tance query efficiently. For the sake of simplicity, in subsequent discussion we assume that the given graph is unweighted for which the weight of each edge is 1 and the distance between two vertices is the minimum hop count between them. We will discuss the necessary adaptations that are needed for a weighted graph at the end of this section.

**Topological Level:** Given a DAG $G$, we use $V(G)$ and $E(G)$ to represent set of vertices and edges of $G$, respectively. The topological level of any vertex $v \in V(G)$, defined as $topo(v)$, is 1 if $v$ has no incoming edge, otherwise it is at least one higher than the topological level of any of $v$'s parents. Mathematically,

$$topo(v) = \begin{cases} \max\limits_{(u,v)\in E(G)} topo(u) + 1, & \text{if } v \text{ has incoming edges} \\ 1, & \text{otherwise} \end{cases}$$

For a vertex $v$, if $topo(v)$ is even, we call $v$ an even-topology vertex, otherwise $v$ is an odd-topology vertex. An edge, $e = (u, v) \in E(G)$, is a single-level edge if $topo(v) - topo(u) = 1$, otherwise it is a multi-level edge. For a DAG $G$, its topological level is the largest value of $topo(v)$ over the vertices in $G$, i.e.:

$$topo(G) = \max_{v\in V(G)} topo(v)$$

**Example:** Consider the DAG $G$ in Figure 1(a). Topological level of vertices, $a, b$, and $c$ is 1, as the vertices have no incoming edge. The topological level of vertex $l$ is 4, as one of the predecessor node of $l$ is $i$, which has a topological level value of 3. $Topo(G)$ is equal to 7, because 7 is the largest topological level value for one of the vertices in $G$. □

Topological compression of a DAG is performed iteratively, such that the compressed output of one iteration is the input of subsequent iteration. For an input DAG $G$, one iteration of topological compression removes all odd-topology vertices from $G$ along with the edges that are incident to the removed vertices. All single-level edges are thus removed, as one of the adjacent vertices of these edges is an odd-topology vertex. A multi-level edge is also removed if at least one of the endpoints of the edge is an odd-topology vertex. As a result of this compression, the topological level of $G$ reduces by half. For the purpose of shortest distance index building, starting from $G = G^0$, we apply this compression process iteratively to generate a sequence of DAGs $G^1, G^2, \cdots, G^t$ such that the topological level number of each successive DAG is half of that of the previous DAG, and the topological level number of the final DAG in this sequence is 1; i.e., $topo(G^{i+1}) = \lfloor topo(G^i)/2 \rfloor$, and $topo(G^t) = 1$, where $t = \lfloor \log_2 topo(G) \rfloor$.

**Example:** Consider the same DAG $G = G^0$ in Figure 1(a). Its topological compression in the first iteration, $G^1$ is shown in Figure 2(a), and in the second iteration, $G^2$ is shown in Figure 2(c). $G^2$ is the last compression state of $G^0$, as topological level of $G^2$ is 1. Note that, in $G^1$, all odd-topology vertices of $G^0$, such as, $a, b, c, h, i, j$, etc. are removed. All single-level edges of $G^0$, such as, $(e, i), (k, n), (p, s)$, etc. are removed. Multi-level edges, such as, $(b, l)$ and $(m, s)$ are also removed. However, there are newly added vertices in $G^1$, such as $b', h', r1$, and $s1$, along with newly added edges, such as, $(b', l)$ and $(e, r1)$. More discussion about these additional vertices and edges are given in the following paragraphs. □

Each iteration of topological compression of a DAG causes loss of information regarding the connectivity among the vertices; for correctly answering distance queries TopCom needs to preserve the connectivity information as the input DAG is being compressed. The preservation process gives rise to additional vertices and edges in $G^1$,

which we have seen in the above example. The connectivity preservation process is discussed in detail below.

The most common information loss is caused by the removal of single-level edges. However, such edges are also easily recoverable from the lastly compressed graph in which the edges were present before their removal. So, TOPCOM does not perform any action for explicit preservation of single-level edges. To preserve the information that is lost due to the removal of multi-level edges, TOPCOM inserts additional even-topology vertices, together with additional edges between the even-topology vertices to prepare the DAG for the compression. The insertion of additional vertices and edges for preserving the information of a removed DAG multi-level edge $e = (u, v)$ is discussed below along with an example given in Figure 1. In this figure, the topological levels are mentioned in rectangular boxes. On the left side we show the original graph, and on the right side we show the modified graph which preserves information that is lost due to compression.

There are four possible cases for an edges $(u, v)$ that is being removed due to topological compression.

**Case 1:** ($topo(u)$ is odd and $topo(v)$ is even). Compression removes the vertex $u$, so we add a **fictitious** vertex $u'$ such that $topo(u') = topo(u) + 1$. Then we remove the multi-level edge $(u, v)$ and replace it with with two edges $(u, u')$ and $(u', v)$. Since topological level number of both $u'$ and $v$ are even, the topological compression does not delete the edge $(u', v)$. For example, consider the multi-level edge $(b, l)$ in figure 1(a), $topo(b) = 1$ (odd), and $topo(l) = 4$ (even). In the modified graph Figure 1(b) this edge is replaced by two edges $(b, b')$ and $(b', l)$, where $b'$ is the fictitious node.

**Case 2:** ($topo(u)$ is even and $topo(v)$ is odd). This case is symmetric to Case 1 as compression removes $v$ instead of $u$. We use a similar approach like Case 1 to handle this case. We create $v_1$, a copy of the vertex $v$ such that $topo(v_1) = topo(v) - 1$ and replace the multi-level edge $(u, v)$ with two edges $(u, v_1)$ and $(v_1, v)$. To distinguish the vertices added in these two cases, the newly added vertex is called **fictitious** for Case 1, and it is called **copied** for Case 2. The justification of such naming will be clarified in latter part of the text. Example of Case 2 in Figure 1(a) is edge $(m, s)$, where $topo(m) = 4$ (even) and $topo(s) = 7$ (odd). In modified graph, we add copied node $s_1$ and replace the original edge with two edges shown in Figure 1(b).

**Case 3:** ($topo(u)$ is odd and $topo(v)$ is odd). In this case we use the combination of above two methods and add two new vertices $u'$ and $v_1$. We set topological level numbering of new vertices as mentioned above. Also we replace multi level edge $(u, v)$ with three different edges $(u, u')$ , $(u', v_1)$, and $(v_1, v)$. Multi level edge $(h, r)$ in Figure 1(a) is an example of this case. As shown in Figure 1(b), we add two new vertices $h'$ and $r_1$ and three new edges, $(h, h')$, $(h', r_1)$, and $(r_1, r)$ after deleting the original edge $(h, r)$. Note that, if $topo(u) = topo(v) - 2$, $topo(u') = topo(v_1)$. In this case, we treat it as Case 1 by adding only $u'$ (but not $v_1$) and following the Case 1. It generates a single-level edge $(u', v)$, which we do not need to handle explicitly.

**Case 4:** ($topo(u)$ is even and $topo(v)$ is even). This is the easiest case as both $u$ and $v$ are not removed by the compression process and we do not make any change in the graph. Also note that the changes in the above three cases convert those cases into this Case 4. For example, applying Case 1 for edge $(b, l)$ in Figure 1 creates new multi-edge $(b', l)$ which is an occurrence of Case 4. Similarly Case 2 creates the Case 4 multi-edge $(m, s_1)$. ■

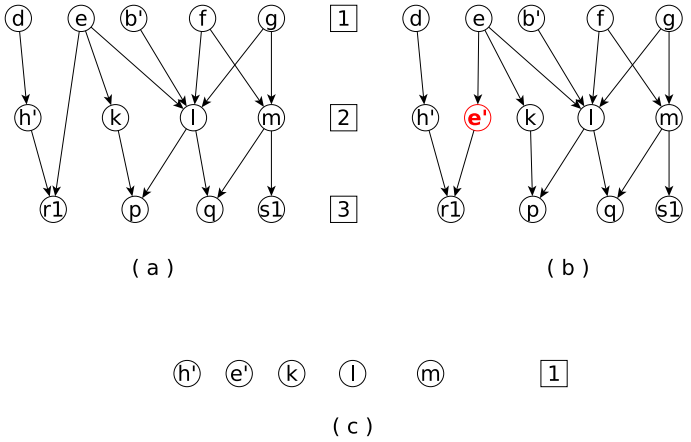



Fig. 2: (a) 1-Compressed Graph $G^1$, (b) Modified 1-compressed Graph $G^1_m$, (c) 2-Compressed Graph $G^2$

**Dummy edges data structure:** We described earlier, we do not need to handle single-level edges separately. However if two continuous single-level edges are removed, we still need to maintain the logical connection between the even-topology vertices. For example, in Figure 1(a) edges $(e, i), (i, k)$, and $(i, l)$ are single-level edges which will be deleted after the first compression iteration because $topo(i) = 3$. Now, information of logical (indirect) connection between $e$ to $k$ and $l$ needs to be maintained, because all three vertices will exist after the compression. To handle this, we add new **dummy edges** $(e, k)$ and $(e, l)$; dummy edges are shown as dotted lines in Figure 1(b). Note that, for any dummy edge $(u, v)$, $topo(v) - topo(u) = 2$ in the current DAG and the edges for which the node-topology difference is higher than 2 are handled by the above 4 multi-level edge cases. For same start and end nodes, if there are multiple dummy edges, TOPCOM considers edge with the smallest distance. To find the dummy edges, we scan through all odd-topology vertices and find their single-level incoming and outgoing edges. We store all these dummy edges along with the corresponding *distance* value in a list called $DummyEdges$ as shown in Figure 1, which we use during the index generation step. For example dummy edge $(d, h')$ has a distance 2 in the Figure 1, then $[(d, h'), 2]$ is stored in $DummyEdges$.

At each compression iteration, we first obtain a modified graph, with fictitious vertices, copied vertices, and dummy edges and then apply compression to obtain the compressed graph of the subsequent iteration. The fictitious vertices, copied vertices, and dummy edges of the modified graph in earlier iteration become regular vertices and edges of the compressed graph in subsequent iteration. The above modification and compression proceeds iteratively until we reach $t$-compressed graph, $G^t$, for which the topological level number is 1. We use $G_m$ to denote the modified uncompressed graph, $G^1_m$ to denote the modified 1-compressed graph, $G^2_m$ to denote the modified 2-compressed graph and so on. For example, Figure 2(a) shows $G^1$ which is obtained by compressing the modified graph $G_m$ in Figure 1(b). Figure 2(b) shows $G^1_m$, the modified 1-compressed graph, and Figure 2(c) shows $G^2$, the 2-compressed graph. We refer the set of all modified compressed graphs as $G^*_m$, i.e. $\{ G^t_m, ..., G^2_m, G^1_m \}$.

### 3.2. Index generation

TOPCOM's index data structure is represented as a table of *key-value* pairs. For each *key*, (vertex) $v$ of the input graph, the *value* contains two lists: (i) outgoing index value

$I_v^{out}$ , which stores shortest distances from $v$ to a set of vertices reachable from $v$; and (ii) incoming index value $I_v^{in}$, which stores shortest distances between $v$ and a set of vertices that can reach $v$. Both the lists contain a collection of tuples, $\langle vertex\_id, distance \rangle$, where *vertex_id* is the id of a vertex other than $v$, and *distance* is the corresponding shortest path distance between $v$ and that vertex.

**Algorithm 1** Outgoing Index *value* Generation

**Input:** $G_m^*$ (set of modified graphs), DummyEdges (set of dummy edges and corresponding distance)
**Output:** $I_*^{out}$ (set of out going indexes for all nodes)

```
1: for all G_m^curr ∈ {G_m^topo(G), ..., G_m^1, G_m} do
2:   O = {u ∈ V(G_m^curr) | topo(u) = odd_number}
3:   for all v ∈ O do
4:     org_v = GETORIGINAL(v)
5:     for all (v, w) ∈ E(G_m^curr) do
6:       org_w = GETORIGINAL(w)
7:       if org_v == org_w then
8:         Continue
9:       end if
10:      if (v, w) == Dummy_Edge then
11:        distance = GETDUMMYDISTANCE(DummyEdges, v, w)
12:      end if
13:      if w == fictitious_vertex then
14:        distance = distance - 1
15:      end if
16:      RECURSIVEINSERT(I_org_v^out, org_w, distance, out)
17:    end for
18:  end for
19: end for
```

At the beginning of the indexing step, for each vertex $v$, TOPCOM initializes $I_v^{out}$ and $I_v^{in}$ with an empty set. It generates index from $G_m^t$ and repeats the process in reverse order of graph compression i.e. from graph $G_m^t$ to $G_m$. In $i$'th iteration of index building, it uses $G_m^{t-i}$ and inserts a set of tuples in $I_v^{out}$ and $I_v^{in}$, only if $v$ is an odd-topology vertex in $G_m^{t-i}$. Thus, during the first iteration, for every odd-topology vertex $v$ of $G_m^{t-1}$, for an incoming edge $(u, v)$ TOPCOM first checks whether $(u, v)$ is in $DummyEdges$ data structure, if so, it inserts $\langle u, d \rangle$ in $I_v^{in}$, where the distance $d$ value is obtained from the $DummyEdges$ data structure. Otherwise, it inserts $\langle u, 1 \rangle$ in $I_v^{in}$. Similarly, for an outgoing edge $(v, w)$ TOPCOM inserts $\langle w, d \rangle$ in $I_v^{out}$, if $(v, w)$ is in $DummyEdges$, otherwise it inserts $\langle w, 1 \rangle$ in $I_v^{out}$. TOPCOM also inserts (Line 16 in Algorithm 1) elements of $I_u^{in}$ and $I_w^{out}$ into $I_v^{in}$ and $I_v^{out}$, respectively, using recursive calls.

Algorithm 1 shows the pseudo-code of the index generation procedure for outgoing index *values* only. An identical piece of code can be used for generating incoming index *values* also, but for that we need to exchange the roles of fictitious and copied vertex, and change the $I_*^{out}$ with $I_*^{in}$ in Line 5-21 (more discussion on this is forthcoming).

As shown in Line 2 of Algorithm 1, TOPCOM first collects all odd-topology vertices in variable $O$ and builds out-indexes for each of these vertices using outgoing edges from these vertices (the edge $(v, w)$ in Line 5 of Algorithm 1). Note that, vertices $v$ and $w$ in $G_m^{curr}$ can be fictitious or copied vertex; TOPCOM uses the subroutine GETORIGINAL()

Table I: Intermediate index generated from the DAG in Figure 2(b)

| ***key*** | Out Index *value* | ***key*** | In Index *value* |
|---|---|---|---|
| **b** | $\{\langle l,1\rangle\}$ | **p** | $\{\langle k,2\rangle,\langle l,2\rangle\}$ |
| **d** | $\{\langle h,1\rangle\}$ | **q** | $\{\langle m,1\rangle,\langle l,2\rangle\}$ |
| **e** | $\{\langle k,2\rangle,\langle l,2\rangle\}$ | **r** | $\{\langle h,1\rangle,\langle e,1\rangle\}$ |
| **f** | $\{\langle l,2\rangle,\langle m,2\rangle\}$ | **s** | $\{\langle m,1\rangle\}$ |
| **g** | $\{\langle l,2\rangle,\langle m,2\rangle\}$ | | |

which returns original vertex corresponding to any fictitious or copied vertex, if necessary (Line 4 and 6). Using the data structure $DummyEdges$ (discussed in section 3.1), it first checks whether the edge $(v,w)$ is a dummy edge (Line 10); if so, it obtains the actual *distance* from the data structure. In case the end-vertex $w$ is a fictitious vertex, TOPCOM decrements the distance value by 1 (Line 14), because for each fictitious vertex, an extra edge with distance 1 is added from the original vertex to the fictitious vertex which has increased the distance value by one. For instance, in the graph in Figure 1, the actual distance from $a$ to $h$ is 2, but the fictitious vertex $h'$ records the distance to be $3$, which should be corrected. On the other hand, if $w$ is a copied vertex, TOPCOM does not make this subtraction, because when a copied vertex is used as destination instead of the original vertex, the distance between the source vertex and the copied vertex correctly reflects the actual distance. For an example, in the same graph, the distance between $e$ and $r$ is 1; when we use the copied vertex $r1$ instead of $r$, distance between $e$ and $r1$ is recorded as 1, which is the correct distance between $e$ and $r$; so no distance correction is needed during the out index building when the destination vertex is a copied vertex. This is the reason why we make a distinction between the fictitious vertices and the copied vertices.

Finallly note that, after generating indexes for each vertex there may be multiple entries for some vertices; from these multiple entries we need to get the smallest value (entry) and remove others. For building incoming index *values*, TOPCOM subtracts 1 for a copied vertex, but does not subtract 1 for a fictitious vertex, as the roles of start and end vertices flip for the incoming index *values*. Below, we give a complete index building example using the vertex $a$ of the graph in Figure 1.

**Example:** We want to find the outgoing index (*value*) for vertex $a$ (*key*) of the graph $G$ in Figure 1(a). $topo(G) = 2$, so we start building index using the graph $G^1_m$, which is shown in Figure 2(b). In the first iteration, TOPCOM builds $I^{out}_d = \{\langle h,1\rangle\}$; the distance value of $1$ comes as follows: TOPCOM uses distance of dummy edge $(d,h')$ that is 2 (Figure 1-II) and then it replaces the fictitious vertex $h'$ with $h$ and obtains a distance of 1 by subtracting 1 from 2 (Line 14). It also inserts the following entries under the *key* $e$; i.e., $I^{out}_e = \{\langle k,2\rangle,\langle l,2\rangle\}$. The resulting indexes after this iteration is presented in Table I; incoming index *values* for keys $b,d,e,f,g$ are empty (not presented in the table) and similarly outgoing index *values* for keys $p,q,r,s$ are empty. For next iteration considering $G_m$, TOPCOM inserts $\langle d,1\rangle$ in $I^{out}_a$; using recursive calls of algorithm 2 (Line 11), this function also inserts $\langle h,2\rangle$ in $I^{out}_a$, recursion stops at $h$ because $I^{out}_h$ is empty (Line 6). Similarly, $\langle e,1\rangle$ and $\langle k,3\rangle,\langle l,3\rangle$ are inserted in $I^{out}_a$ recursively from $I^{out}_e$. At the end of the algorithm 1 we remove duplicate entries from indexes. For example, incoming index for key $s$ has two entries for vertex $m$, $\langle m,1\rangle$ and $\langle m,2\rangle$, one corresponding to edge $(m,s1)$ in $G^1_m$ and the other is a recursive result from $q$ to $s$ in $G_m$. TOPCOM considers $\langle m,1\rangle$ and discards the other entry from $I^{in}_s$. For the graph in Figure 1(a), corresponding indexes are presented in Table II.

Table II: Index for the DAG in Figure 1

| ***key*** | Out Index *value* | In Index *value* |
|---|---|---|
| **a** | $\{\langle d,1\rangle, \langle e,1\rangle, \langle h,2\rangle, \langle k,3\rangle, \langle l,3\rangle\}$ | $\emptyset$ |
| **b** | $\{\langle l,1\rangle, \langle f,1\rangle, \langle m,3\rangle\}$ | $\emptyset$ |
| **c** | $\{\langle f,1\rangle, \langle g,1\rangle, \langle l,3\rangle, \langle m,3\rangle\}$ | $\emptyset$ |
| **d** | $\{\langle h,1\rangle\}$ | $\emptyset$ |
| **e** | $\{\langle k,2\rangle, \langle l,2\rangle\}$ | $\emptyset$ |
| **f** | $\{\langle l,2\rangle, \langle m,2\rangle\}$ | $\emptyset$ |
| **g** | $\{\langle l,2\rangle, \langle m,2\rangle\}$ | $\emptyset$ |
| **h** | $\emptyset$ | $\{\langle d,1\rangle\}$ |
| **i** | $\{\langle k,1\rangle, \langle l,1\rangle\}$ | $\{\langle e,1\rangle\}$ |
| **j** | $\{\langle l,1\rangle, \langle m,1\rangle\}$ | $\{\langle f,1\rangle, \langle g,1\rangle\}$ |
| **n** | $\{\langle p,1\rangle\}$ | $\{\langle k,1\rangle\}$ |
| **o** | $\{\langle p,1\rangle\}$ | $\{\langle l,1\rangle\}$ |
| **p** | $\emptyset$ | $\{\langle k,2\rangle, \langle l,2\rangle\}$ |
| **q** | $\emptyset$ | $\{\langle m,1\rangle, \langle l,2\rangle\}$ |
| **r** | $\emptyset$ | $\{\langle h,1\rangle, \langle e,1\rangle, \langle p,1\rangle, \langle k,3\rangle, \langle l,3\rangle\}$ |
| **s** | $\emptyset$ | $\{\langle m,1\rangle, \langle p,1\rangle, \langle k,3\rangle, \langle l,3\rangle, \langle q,1\rangle\}$ |

**Algorithm 2** RECURSIVEINSERT($I_v^{io}, a, distance, in\_or\_out$)

1: **if** $in\_or\_out == in$ **then**
2: $I_a^{io} = I_a^{in}$
3: **else**
4: $I_a^{io} = I_a^{out}$
5: **end if**
6: **if** $I_a^{io} == \emptyset$ **then**
7: $add\_tuple(I_v^{io}, a, distance)$
8: **else**
9: $add\_tuple(I_v^{io}, a, distance)$
10: **for all** $(x, dist) \in I_a^{io}$ **do**
11: RECURSIVEINSERT($I_v^{io}$, $x$, $distance + dist$, $in\_or\_out$)
12: **end for**
13: **end if**

### 3.3. Index for weighted graph

For weighted graph, TOPCOM makes some minor changes in the above algorithm. First, distance values are stored both in the indexes and in the $DummyEdge$ data structure. Many of these distances are implicitly 1 for unweighted graph, which is not true for weighted graph, so, for the latter TOPCOM stores the distance explicitly. Also, it ensures that the distance value between fictitious (or copied) vertices and an original vertex is one, so that the Algorithm 1 works as it is.

### 3.4. Query processing

For query processing, TOPCOM uses the distance indexes that is built during the indexing stage. For a given distance query from $u$ to $v$, i.e. to compute $\delta(u, v)$, TOPCOM intersects outgoing index *value* of *key* $u$ i.e. $I_u^{out}$ and incoming index *value* of *key* $v$ i.e. $I_v^{in}$ and finds common *vertex_id* in $I_u^{out}$ and $I_v^{in}$, along with the *distance* values. To cover the cases, when $v$ is in the outgoing index *value* of $u$, or $u$ is in the incoming index *value* of $v$, the tuples $\langle u, 0\rangle$ and $\langle v, 0\rangle$ are also added in $I_u^{out}$ and $I_v^{in}$ respectively

and then the intersection set of the indexes is found. If the intersection set size is 0, there is no path from $u$ to $v$ and hence the distance is infinity. Otherwise, the distance is simply the sum of the *distances* from $u$ to *vertex_id* and *vertex_id* to $v$. If multiple paths exist, we take the one that has the smallest distance value.

**Example:** We want to find $\delta(a, s)$ in Figure 1. From table II, $I_a^{out} \cap I_s^{in} = \{k, l\}$. Now, we need to sum up the corresponding *distance* values, that gives $\{\langle k, 6\rangle, \langle l, 6\rangle\}$. Now we need to find smallest distance value; in this case both the values are same, hence we can provide any one as a result.

### 3.5. Theoretical proofs for correctness

In this section, we prove the correctness of TOPCOM, through the claim that TOPCOM's index is based on 2-hop covers of the shortest distance in a graph and method described in Section 3.4 gives correct shortest distance value. For shortest path, such a cover is a collection $S$ of shortest paths such that for every two vertices $u$ and $v$, there is a shortest path from $u$ to $v$ that is a concatenation of atmost two paths from $S$. [Cohen et al. 2002]. That is, shortest path from $u$ to $v$ is stored in $S$ or there is an intermediate node $x$ such that shortest paths from $u$ to $x$ and from $x$ to $v$ are stored in $S$. For TOPCOM's index also, the shortest distance from any node $u$ to node $v$ is the 2-hop cover such that the index itself has shortest distance value from $u$ to $v$ or there is an intermediate node $x$ which would be present in both $I_u^{out}$ and $I_v^{in}$.

**Example:** In DAG $G$ in Figure 1(a) to find distance from $a$ to $s$, we need to check the outgoing index value for vertex $a$ and the incoming index value for vertex $s$ in Table II. This gives us two possible shortest paths passing through intermediate node $k$ or $l$, because distance in both cases is same. Thus, there can be multiple shortest paths however, atmost one intermediate node in the index.

In the Theorem 3.3, we try to identify the topological layer of an intermediate node $x$. We identify a unique topological level for each pair of $u$ and $v$, which tells there is atmost one intermediate node in a shortest path from $u$ to $v$ because in DAG there cannot be a directed edge within topological layer. We begin with the following lemmas, which will be useful for constructing the proof of the theorem.

LEMMA 3.1. *In $G_m$, if a node $u$ is at topological level $2^i$, it will be at topological level $1$ in $G^i$.*

PROOF. TOPCOM compression method removes all odd-topology nodes and carries over nodes from the even topological levels to the next compression iteration. Thus any node from an even topological level $2x$ in some compressed graph will be at topological level $x$ in the compressed graph of next iteration. Say, the node $u$ is at topological level $2^i$ in $G_m$, then it will be at topology level $2^{i-1}$ in $G^1$. Since $2^{i-1}$ is also even, no fictitious or copied vertex will be added for $u$, and in $G_m^1$, it will remain at $2^{i-1}$ level. In the next compression iteration, $u$ will simply be moved to $2^{i-2}$ level in $G^2$ and so on. Hence, it will be at level $2^{i-i} = 2^0 = 1$ level in $G^i$ graph. □

**Example** In the graph $G_m$ shown in Figure 1(b), the node $d$ is at topological level $2$ and the node $k$ is at topological level $4$. In $G^1$ shown in Figure 2(a) the node $d$ is at topological level $1$; similarly, in $G^2$ shown in Figure 2(c), the node $k$ is at topological level $1$.

LEMMA 3.2. *In* TOPCOM*'s index, for all keys, the values contain vertices, which are only from even topological level in the modified DAG $G_m$.*

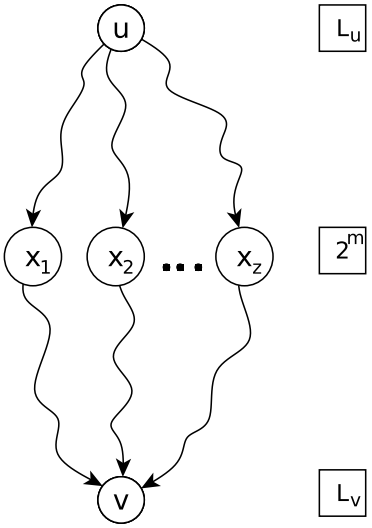



Fig. 3: Shortest path from $u$ to $v$ passing through $x$

PROOF. As per Line 2 of Algorithm 1, TOPCOM's index *keys* are nodes from only an odd topological level, and the *values* of index are built using the incident edges of those key nodes. In the modified graph $G_m$, all the edges from/to an odd topology vertex connects with an even topology vertex, through the use of fictitious/copied nodes (if needed). Hence, if any node in DAG $G_m$ is at an odd topological level, it cannot be included as an index *value*. Additionally when we compress $G_m^i$ to get $G^{i+1}$, we only include nodes from even topological levels, hence nodes from odd levels will never be included as a *value* for index building at compressed levels also. □

**Example:** See the completely built index of the graph $G$ in Figure 1(a) as shown in Table II. The nodes that appear as *values* are $\{d, e, b'(b), f, g, h'(h), k, l, m, r_1(r), p, q, s_1(s)\}$. All of these are from the even topology nodes in $G_m$ as shown in Figure 1(b).

THEOREM 3.3. *For finding shortest distance from $u$ to $v$, assume that $u$ has topological level number $L_u$ and $v$ has topological level number $L_v$ in $G_m$. We define*

$$n = \underset{i}{\text{argmax}}(L_u \leq 2^i \leq L_v) \tag{1}$$

*Now, if there is a shortest path from $u$ to $v$, for each shortest path, exclusively, one of the following is true.*

*Case 1: No intermediate node $x$ i.e. $I_u^{out}$ includes $v$ or $I_v^{in}$ includes $u$.*

*Case 2: There is an intermediate node $x$, and*

$$topo(x) = 2^n + C$$

*for some constant offset C.*

PROOF. We prove this theorem using mathematical induction on $n$.

**Base case:** $n = 1$. If there is a direct edge from $u$ to $v$ then *case 1* is true because if $L_u = 2$ then $I_v^{in}$ includes $u$ or if $L_v = 2$ then $I_u^{out}$ includes $v$. If there is an intermediate

node $x$ then $L_u = 1$ and $L_v = 3$, hence $topo(x) = 2$ that shows *case 2* is true. From Lemma 3.2 both $I_u^{out}$ and $I_v^{in}$ include the node $x$ if there is a path from $u$ to $v$. In this case constant offset *C* would be zero.

**Induction hypothesis:** Here we assume that for $n = d$ given theorem is true.

**Induction step:** We want to prove, for $n = d + 1$ given theorem is true.

If there is no intermediate node $x$ then *case 1* is true. Hence, we discuss the only scenario where there is an intermediate node $x$ and we want to show that $x$ is in both $I_u^{out}$ and $I_v^{in}$. We sub-divide the proof for zero and non-zero values of constant offset *C*.

*Constant offset C is zero:*

If there are $2^{d+1}$ levels, then compression step would be conducted at least one more time than $2^d$ levels. From Lemma 3.1 at the $d$'th step of compression, nodes at topological level $2^d$ in graph $G_m$ are at $1^{st}$ topological level in $G^d$ and nodes from topological level $2^{d+1}$ would be at $2^{nd}$ topological level.

Hence, TOPCOM will build index for *keys* (nodes) from topological level $2^d$ in graph $G_m$, and those index *values* include nodes from topological level $2^{d+1}$. As our method recursively includes already built index *values*, the nodes from topological level $2^{d+1}$ would be recursively included to corresponding outgoing index *values* for *keys* at lower compression levels. Hence, if $topo(x) = 2^{d+1}$ then it is present in outgoing index value of $u$.

The similar argument works for incoming index of $v$.

*Constant offset C is non-zero:*

If we cannot find $n$ that satisfies equation 1 then constant offset *C* is non-zero. In this case offset can be calculated as :

$$\begin{aligned} C &= 2^{n_{low}} \\ where, \qquad n_{low} &= \underset{i}{\operatorname{argmax}}(2^i < L_u) \end{aligned} \tag{2}$$

Now, we define modified topological level number of $u$ is $L_u^m$, where $L_u^m = L_u - C$ and similarly modified topological level number of $v$ is $L_v^m = L_v - C$. We use $L_u^m$ and $L_v^m$ in equation 1 to get $n$

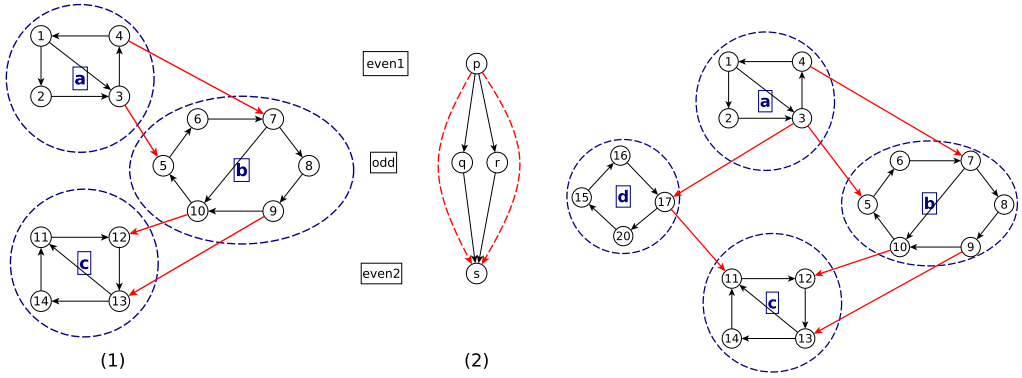
$$n = \underset{i}{\operatorname{argmax}}(L_u^m \leq 2^i \leq L_v^m)$$

Now, with $topo(x) = 2^n + C$, argument works similarly as zero offset.

□

Example of non-zero offset *C*: In Figure 1(b), we want to know the shortest distance from $n$ to $r$ where corresponding topological levels are $L_n = 5$ and $L_r = 7$ respectively. For this, we can not find any $n$ that satisfies the equation 1. From equation 2, we can calculate $n_{low} = 2$, using which modified topological levels $L_n^m = 1(5 - 2^2)$ and $L_r^m = 3$ can be obtained. From $L_n^m$ and $L_r^m$ we get $m = 1$. Now, $2^1 + 2^2 = 6$ is the topological level of intermediate node $p$, which is present in both $I_n^{out}$ and $I_r^{in}$ (Table II). If we look carefully $L_n^m$ and $L_r^m$ is a base case in the mathematical induction proof of Theorem 3.3, and $L_n(5)$, $L_r(7)$ with offset *C* behave exactly the same as the base case.

**Note:** If there is no node from topological level $2^n$ in the shortest path from $u$ to $v$, then there must be one multilevel edge which skips that level. For a node incident to that multilevel edge, at some step of the compression, we need to prepare fictitious/copied node. That new fictitious/copied node works as a node from topological




(a) (1)Example for Distance within Middle DAG node of $G_d$ and (2)Example of multiple dummy edges
(b) Example of merging multiple dummy DAG edges in modified $G_d$

Fig. 4: Dummy edge handling

level $2^n$ and will be included in both $I_u^{out}$ and $I_v^{in}$. Thus, theorem works fine for this case.

For example, as depicted in Figure 1(b), shortest path from $a$ to $r$ doesn't pass through any node from topological level $4(2^2)$ in $G_m$, but it has a multilevel edge $(e, r_1)$. In $G_m^2$ (Figure 2(b)), this edge causes a fictitious node $e'$ at topological level $2$ which is (logically) topological level $4$ in $G_m$. The resulting index in Table II shows that, the node $e'(e)$ is included as a *value* in the incoming index of $r$ ($I_r^{in}$) and also included in $I_a^{out}$.

## 4. INDEXING FOR GENERAL DIRECTED GRAPH

Any directed graph $G$ can be converted to a Directed Acyclic Graph (DAG) $G_d$, by considering each strongly connected component (SCC) of $G$ as a node of $G_d$. Thus in DAG, the edges within a SCC are collapsed within the corresponding node. However, if an edge in $G$ connects two vertices from two distinct SCCs, in $G_d$ those SCCs are connected by a DAG edge. To build the shortest path index for a general directed graph, TOPCOM first uses Tarjan's algorithm [Tarjan 1972] to convert $G$ to a DAG $G_d$ by finding all SCCs of $G$. It also maintains a necessary data structure that keeps the mapping from a DAG node to a set of graph vertices, and vice-versa. We call this a parent-child mapping, i.e., a DAG node is the parent of graph vertices which are part of the corresponding SCC. A DAG edge connects two vertices, one from a distinct SCC. We call such vertices terminal vertices. A single DAG edge between a pair of SCCs may encapsulate multiple paths (one edge or multiple edges) of Graph $G$ such that the end vertices of these paths are terminal vertices in those pair of SCCs. TOPCOM's DAG edge data structure contains a set of tuples, each representing one of these paths. A tuple has three elements: node-id of start terminal vertex, node-id of end terminal vertex, and the distance between these two vertices in $G$. For example consider Figure 4a(1), a DAG edge $(a, b)$ stores $\{(3, 5, 1), (4, 7, 1), (3, 7, 2), (4, 5, 3)\}$, first two tuples represent a single-edge path, but the last two represent multi-edge paths. 3, 4 are terminal vertices of DAG node $a$, and 5,7 are terminal vertices of DAG node $b$. For each tuple, the third field stores the shortest distance between the pair of terminal vertices in the first two fields of that tuple. To compute distance between an arbitrary pair of vertices within an SCC, TOPCOM also pre-computes all-pair shortest paths among all

Table III: Real world datasets and basic information

| Name | \| V \| | \| E \| | AD | MD | $\| V_{DAG} \|$ | $\| E_{DAG} \|$ | $AD_{DAG}$ | $MD_{DAG}$ | Largest SCC |
|---|---|---|---|---|---|---|---|---|---|
| AS_Caida | 26,374 | 2,304,095 | 87.36 | 2,205,805 | 26,358 | 48,958 | 1.86 | 2606 | 8 |
| Email_Eu | 265,214 | 420,045 | 1.58 | 7,636 | 231,000 | 223,004 | 0.97 | 168,815 | 34,203 |
| Epinion | 49,289 | 487,183 | 9.88 | 2,631 | 16,264 | 16,497 | 1.01 | 15,789 | 32,417 |
| Gnutella09 | 8,114 | 26,013 | 3.21 | 102 | 5,491 | 6,495 | 1.18 | 5,147 | 2,624 |
| Gnutella31 | 62,586 | 147,892 | 2.36 | 95 | 48,438 | 55,349 | 1.14 | 43,928 | 14,149 |
| WikiVote | 7,116 | 103,689 | 14.57 | 1,167 | 5,817 | 19,540 | 3.36 | 4869 | 1,300 |

graph nodes belonging to a single SCC and store them in shortest path index. In real-life directed networks, the size of SCCs are generally not very large, so storing all-pair distances within a SCC in the shortest path index is feasible.

### 4.1. Distance for dummy edges:

For an unweighted DAG two consecutive edges yield a distance value of 2, but for DAG which is a compressed representation of a general unweighted directed graph, two consecutive DAG edges may constitute an arbitrary distance value. This is due to the fact that the shortest path may visit a large number of vertices which are part of the start, middle, and end SCC. For an example, see Figure 4a(1); in this figure, the rectangles "even1", "odd" and "even2" are the topological level numbers; $a$, $b$ and $c$ are the DAG nodes (ellipses), and nodes with numeric ids are nodes of $G$. Two consecutive DAG edges are $(a, b)$ and $(b, c)$ connecting SCCs $a$, $b$ and SCCs $b$, $c$, respectively. The shortest distance between $a$ and $c$ depends on the terminal nodes of $a$ and $c$ that are being used. If terminal node of $a$ is $3$ and terminal node of $c$ is 13, the distance is 5, following the path $3 - -4 - -7 - -10 - -12 - -13$. In this path, besides the distance 2 over the DAG edges, there are three within-SCC edges, one in each of SCCs. Thus, the total distance for a dummy edge is the sum of (i) distance from a terminal vertex of staring SCC to a terminal vertex in the middle SCC, (ii) distance between a pair of terminal vertices in the middle SCC, and (iii) distance from a terminal node in the middle SCC to a terminal node in the end SCC. To account for this, TOPCOM computes the dummy edge distance by considering all possible combination of terminal nodes in each SCCs.

**Example:** Say, TOPCOM wants to find dummy DAG edge $a$ to $c$ which would be set of tuple $\{(3, 12, *), (3, 13, *), (4, 12, *), (4, 13, *)\}$, (all sources to all destinations) where $*$ represents the shortest *distance* values that it needs to find. To find the distance from node $3$ to $13$, it finds the distance for all combinations of terminal nodes in the middle SCCs and takes the minimum. From starting SCC to middle SCC ($\delta(3, 5) = 1$ and $\delta(3, 7) = 2$), within middle SCC ($\delta(5, 9) = 4$, $\delta(5, 10) = 3$, $\delta(7, 9) = 2$ and $\delta(7, 10) = 1$) and finally from middle SCC to end SCC ($\delta(9, 13) = 1$, $\delta(10, 13) = 2$). In this example, $\delta(3, 7) + \delta(7, 10) + \delta(10, 13)$ gives the minimum value $5$ which generates the tuple $(3, 13, 5)$. Distance for all other tuples are also calculated similarly.

**Multiple dummy edges:** Another issue is, there could be multiple dummy edges having same starting and ending DAG nodes as shown in Figures 4a(2). If the original graph itself is a DAG, TOPCOM considers the dummy edge with the lowest *distance*. But for converted DAG $G_d$ applying this solution is more complex, because distance within middle SCC can be different for different SCCs. For this, we need to merge all possible tuples of all dummy edges, and recalculate the distances by taking the minimum distance from the merged set of tuples.

Table IV: Average Query Time for DAG ($\mu$s)

| Name | **TopCom** | **IS-Label** | **Bi-Djk** | **TreeMap** [3] |
|---|---|---|---|---|
| AS_Caida | 0.1036 | 0.2237 | 24.75 | 0.2471 |
| Email_Eu | 0.1059 | 0.3865 | 1657.46 | 0.2674 |
| Epinion | 0.0360 | 0.2388 | 14.83 | 0.1722 |
| Gnutella09 | 0.0345 | 0.3292 | 7.27 | 0.115 |
| Gnutella31 | 0.0752 | 0.2095 | 50.74 | 0.254 |
| WikiVote | 0.1551 | 0.3494 | 43.11 | 0.2131 |

**Example** in Figure 4b we extend the example of Figure 4a(1) with one more DAG node $d$, which also connects node $a$ to node $c$. Dummy edge through the middle node $d$ is $\{(3,11,2),(4,11,4),(3,12,3),(4,12,5),(3,13,4),(4,13,6)\}$, and through the middle node $b$ is $\{(3,11,6),(4,11,5),(3,12,4),(4,12,3),(3,13,5),(4,13,4)\}$. For dummy edge $(a,c)$, we combine both the sets of tuples and obtain the smallest distance. Thus the final representation of dummy edge $(a,c)$ is the following: $\{(3,11,2),(4,11,4),(3,12,3),(4,12,3),(3,13,4),(4,13,4)\}$.

### 4.2. Modification in index and query processing

TOPCOM Index for general graph stores the bidirectional mapping between the DAG nodes and the vertices of the input graph. For every DAG edge (and also for DAG dummy edges), it stores the set of tuple based representation that we have discussed in the above subsection. It also stores all pair distance between each of the vertices within an SCC. Above all, it prepares and stores the 2-hop cover DAG indexes for the DAG representation of the input graph using the methodologies that we discussed in Section 3.

For query processing, given a query $(u,v)$, TOPCOM first identifies the corresponding SSE nodes in the DAG using the bidirectional map. Say, these SSEs are $s_u$ and $s_v$, respectively. If $s_u = s_v$, TOPCOM simply uses the within SSE all-pair index and return the distance between $u$ and $v$. Otherwise, it first finds the set of out-terminal SSE nodes of $s_u$ (say, $X$), and in-terminal SSE nodes of $s_v$ (say, $Y$). Then it uses the 2-hop cover indexing for finding the shortest path distance between each pair of nodes—one from $X$, and the other from $Y$. It also considers within SCC distances in three SCCs:
**Starting DAG node:** Distance from starting node of the query to start terminal node of the DAG node. For example consider figure 4a(1), where query is $\delta(1,14)$. Now all outgoing edges from DAG $a$ are from nodes $3$ and $4$, hence we need to get distances from $1$ to $3$ and $4$ i.e. $\delta(1,3)=1$ and $\delta(1,4)=2$.
**Middle DAG node:** If there is a middle node (from the 2-hop cover index) then distance from incoming edge terminal node to outgoing edge terminal node within middle DAG node needs to be calculated. In our example suppose $b$ is middle node, then we need distance from $5$ to $9$ and $10$, i.e. $\delta(5,9)=4$, $\delta(5,10)=3$ and similar for node 7.
**Ending DAG node:** Distance from end terminal node to ending node of the query within the DAG node.:w That means in our example distance from $12$ and $13$ to $14$ i.e. $\delta(12,14)=2$ and $\delta(13,14)=1$.

Here again our task is to get minimum distance among all, and we use the similar strategy as section 4.1, which is to minimize the summation of above three distances along with edge distances.

As we can see, TOPCOM's principle indexing process works with DAG and it performs well on real world datasets (Table V). One reason is, real world complex graph

[3]Unweighted Graph results

Table V: Average Query Time for General Graph ($\mu$s)

| Name | **TopCom** | **IS-Label** | **Bi-Djk** | **TreeMap** [3] | **TopCom** [4] |
|---|---|---|---|---|---|
| AS_Caida | 0.26282 | 0.2229 | 25.9614 | 0.3873 | 0.26044 |
| Email_Eu | 12.4708 | 21.98527 | 1482.41 | 0.8102 | 10.45136 |
| Epinion | 34.6582 | 2114.02 | 3570.8667 | 5.727 | 33.1907 |
| Gnutella09 | 1.3405 | 8.0429 | 110.6521 | 1.6255 | 1.29084 |
| Gnutella31 | 2.46202 | 13.9999 | 299.423 | 5.804 | 2.44672 |
| WikiVote | 18.3593 | 23.72954 | 183.4193 | 8.371 | 19.06744 |

becomes less complex when converted as DAG. For example, average degree of DAG $AD_{DAG}$ (Table III) are always smaller than $AD$, mostly an order of magnitude smaller and specifically for $AS\_Caida$ average degree reduced to $1.86$ from $87.36$, which is a decrease of almost two orders of magnitude. However because of DAG, we also need to handle a challenging task, i.e. maintaining distance information within SCC for each DAG node. To keep this information, most common ways are to maintain distance matrix or to keep set of edges and calculate distance at run time. Both of these methods have their own pros and cons; keeping distance matrix is the fastest access method but the size of SCC leads to space limitation i.e. we need space for $O(n^2)$ elements and for huge SCC this may be a notable problem. On the other hand keeping set of edges is a memory efficient way, however all distance finding algorithms are polynomial time in terms of $|V|$ and $|E|$; and for huge SCC, finding distance between nodes at run time would be much slower. This represents a well known phenomenon in Computer Science called space-time trade-off. Here for our task, time gets priority over space, hence we selected first method, where we are maintaining distance matrix for each DAG node. For large SCC, this may take high memory, however we observed that our index is still not very large for contemporary machine.

### 4.3. Correctness revisited

In this Section 4, we explain how to adopt the proposed indexing method for a general directed graph. For that, first we convert a general directed graph into DAG and then build index on the DAG. We describe the methods to maintain the information at both steps.

We should be able to calculate shortest distance from one node to any other node within the same DAG node. When we convert a general directed graph to DAG, we maintain this information by creating appropriate data structures during conversion (Section 4).

The index generation method for DAG is further divided into two steps: 1) Topological Compression and 2) Index Generation. We need to maintain information only during the first step, because the index generation step only builds index from the graphs generated in the topological compression (first) step. The topological compression step is described in Section 3.1, where DAG is compressed iteratively by removing all odd-topology vertices and incident vertices. This compression process maintains loss of information using dummy edges, to keep the correct information we explicitly handle distance information for dummy edges as explained in Section 4.1.

Lastly, as the structure of a converted DAG is different, we need to handle the queries a little differently. We have explained the modification of query processing in Section 4.2. Hence, all the required logical modifications are handled and TOPCOM maintains correctness for general directed graph.

[4]Unweighted Graph: Avg. over 5 times execution for 10K queries

## 5. EXPERIMENTAL EVALUATION

We compare performance of TOPCOM with two of the recent methods (IS-Label and TreeMap) for answering distance query. We also compare TOPCOM with baseline method Bidirectional Dijkstra's algorithm, which is one of the fastest online methods for single source shortest distance queries. For both IS-Label and TreeMap, codes are provided by their authors. For these experiments we use a machine with Intel 2.4 GHz processor, 8 GB RAM and Ubuntu 14.04 LTS OS. In [Xiang 2014] the author has claimed that TreeMap works for weighted directed graphs, however we are provided with the code of unweighted version for TreeMap, hence all comparisons with TreeMap are for unweighted graphs. Additionally as Y. Xiang mentioned in the paper, TreeMap needs huge memory if tree width is above threshold (1000). The only dataset we are able to run using above machine is WikiVote. Hence for comparison with TreeMap, we used machine with AMD 2.3 GHz processor, 132 GB RAM and Red Hat Enterprise Server Release 6.6 OS. We also perform comparison to IS-Lable using same machine for two datasets (Email_Eu and Epinion). Using synthetic graphs of different sizes and degrees, we show that TreeMap is not scalable for higher degree graphs. To generate these synthetic graphs we use python package networkx (procedure name, FAST_GNP_RANDOM_GRAPH()).

### 5.1. Datasets

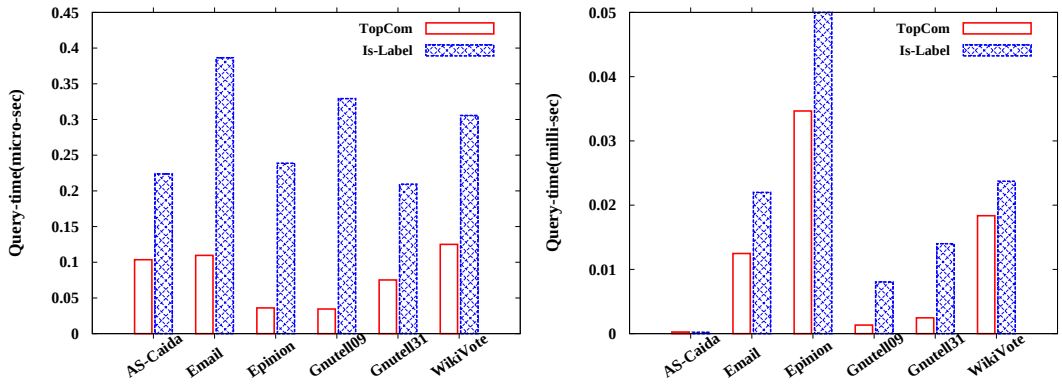
Here for our experiments, we used seven real world datasets (Table III) from different domains to show wide applicability of TOPCOM . $|V|$ and $|E|$ are the number of vertices and the number of edges respectively. Similarly $|V_{DAG}|$ and $|E_{DAG}|$ are the number of vertices and the edges in the DAG of the corresponding graph. ***AD*** and $AD_{DAG}$ are average degree values for the graph and its DAG counterpart, respectively. ***MD*** and $MD_{DAG}$ are maximum degrees i.e. maximum in or out degree in the graph and its DAG, respectively. ***Largest SCC*** is a size of the biggest DAG node which encapsulate the maximum number of input graph nodes.

We collected all datasets from SNAP (Stanford Network Analysis Project) web page[5] except $Epinion$ trust network dataset, which we collected from [Massa and Avesani 2006]. $AS\_Caida$ is a business relationship network and $Email\_Eu$ is a snapshot of an email network generated by European Research Institute. $Epinion$ dataset is a trust network generated from social network users, it represents which user trusts whom. $Gnutella$ is a peer-to-peer file sharing network where $Gnutella09$ is a snapshot of the network on 9th August 2002 and $Gnutella31$ is a snapshot of the same network on 31st August 2002. $WikiVote$ is a network generated from Wikipedia admin voting history data.

### 5.2. Results and Discussion

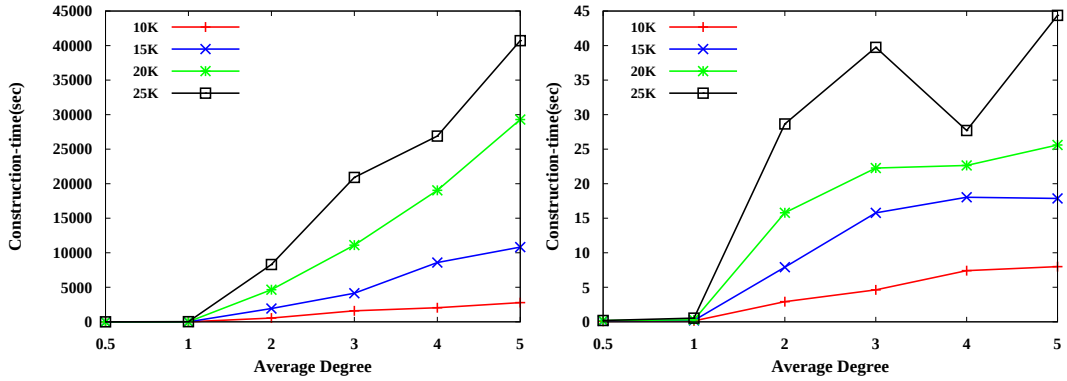
AS per expectation for DAG TOPCOM outperforms IS-Label method for all datasets depicted in figure 5a. Results are average query time over 10 times execution in microsecond ($\mu$s), where each method calculated 10K random queries in every execution. For more detailed comparison, if we look at table IV, we can see that for datasets $Epinion, Gnutella09$ and $Gnutella31$ TOPCOM outperforms IS-Label and TreeMap by an order of magnitude. For other datasets also TOPCOM performs 2-3 times better than both of the competing methods. If we look at the Bi-Dijkstra results, TOPCOM performs multiple orders of magnitude better for all datasets. In figure 5b results for general graphs are plotted, which clearly demonstrate superiority of TOPCOM over IS-Label for general graph. Here also we used Average Query time over 10 times ex-

[5]http://snap.stanford.edu/data/index.html




(a) For DAG ($\mu$s) (b) For General Graph (ms)

Fig. 5: Average Query time comparison




(a) TreeMap Results on Synthetic Graphs (b) TopCom Results on Synthetic Graphs

Fig. 6: Index Building time for Synthetic graphs

ecution with each run calculating 10K random queries. Now if we look at table V for dataset $AS\_Caida$ IS-Label performs better than TOPCOM however the difference is $0.00003989\ ms$ which is really very small. On the other hand TOPCOM outperforms IS-Label for all other datasets. For $Gnutella31$ dataset TOPCOM performs an order of magnitude better than IS-Label and surprisingly IS-Label performs really poor on $Epinion$ dataset such that TOPCOM outperforms IS-Label by two orders of magnitude. For $Guntella09$ TOPCOM performs almost 7 times better and for remaining datasets TOPCOM performs almost two times better than IS-Label. Here once again TOPCOM is multiple orders of magnitude faster than Bi-Dijkstra for all datasets.

Table V shows result of TreeMap and TOPCOM comparisons on unweighed graph. It is clear that TOPCOM is competitively better in $AS\_cadia$, $Gnutella09$ and $Gnutella31$ datasets, but TreeMap performs an order of magnitude better for other three datasets. However, when we run the TreeMap for building index it took hours to build the index for some datasets, for example average index building time for $Epinion$ dataset is more than 9 hours, while $Gnutella31$ takes more than 26 hours. We believe one of the reasons is a bigger graph with higher average degree. To find out the actual cause,

we generated synthetic graphs of different sizes (10000-25000) and degrees (0.5-5) and tried to build indexes using TreeMap. In figure 6a the index building time is shown in seconds, after degree 1 all graphs started taking higher time and for bigger graphs the slope of the curve is very large. We compare construction time of TOPCOM for the same set of synthetic graphs shown in figure 6b, and as shown TOPCOM hardly takes few seconds for index construction. Highest time taken by TOPCOM is 44 sec for 25K nodes graph with average degree 5, which is almost thousand times faster compared to TreeMap. From this we can easily conclude that it is difficult for TreeMap to scale for higher degree graphs.

## 6. CONCLUSIONS

In this paper we proposed TOPCOM : a unique indexing method to answer distance query for directed real-world graphs. This method uses topological ordering property of DAG and describes a novel method for distance preserving compression of DAG. We compared TOPCOM with IS-Label and found the our method performs better than IS-Label for both weighted DAG and weighted general graph. We strongly believe our method should perform similar or better for unweighed graphs, because we store distance information in label irrespective of weighted/unweighted edges. We do not compare TOPCOM with other recent methods such as HCL [Jin et al. 2012b] and state-of-art 2-Hop [Cohen et al. 2002], because Fu et al. have compared IS-Label with HCL and proved superiority of IS-Label in [Fu et al. 2013]. We also compare TOPCOM with the recent TreeMap method, which performs better for some datasets, however, we show that this method is not scalable for huge graphs with higher degree. We plan to study further to build index for dynamic large graphs that can answer exact distance query in an acceptable time.

## REFERENCES


Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2011. A Hub-based Labeling Algorithm for Shortest Paths in Road Networks. In *International Conference on Experimental Algorithms (SEA'11)*. 230–241.

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2012. Hierarchical Hub Labelings for Shortest Paths. In *Annual European Conference on Algorithms (ESA'12)*. 24–35.

Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast Exact Shortest-path Distance Queries on Large Networks by Pruned Landmark Labeling. In *ACM SIGMOD*. 349–360.

Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution *(SIGMOD)*. 44–54.

Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. 2010. Combining Hierarchical and Goal-directed Speed-up Techniques for Dijkstra's Algorithm. *J. Exp. Algorithmics* 15, Article 2.3 (March 2010), 1.21 pages.

Richard Bellman. 1958. On a Routing Problem. *Quart. Appl. Math.* 16 (1958), 87–90.

James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. TF-Label: A Topological-folding Labeling Scheme for Reachability Querying in a Large Graph *(SIGMOD)*. 193–204.

James Cheng, Yiping Ke, Shumo Chu, and Carter Cheng. 2012. Efficient Processing of Distance Queries in Large Graphs: A Vertex Cover Approach. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 457–468.

Jiefeng Cheng and Jeffrey Xu Yu. 2009. On-line Exact Shortest Distance Query Processing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*. ACM, New York, NY, USA, 481–492.

Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2002. Reachability and Distance Queries via 2-hop Labels. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*. 937–946.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

VachikS. Dave and MohammadAl Hasan. 2015. TopCom: Index for Shortest Distance Query in Directed Graph. In *Database and Expert Systems Applications*. Lecture Notes in Computer Science, Vol. 9261. Springer International Publishing, 471–480.

A. Erdem Sariyuce, K. Kaya, E. Saule, and U.V. Catalyurek. 2013. Incremental Algorithms for Network Management and Analysis based on Closeness Centrality. *ArXiv e-prints* (2013).

Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 2013. IS-Label: An Independent-set Based Labeling Scheme for Point-to-point Distance Querying. *Proc. VLDB Endow.* 6 (2013), 457–468.

Jun Gao, Ruoming Jin, Jiashuai Zhou, Jeffrey Xu Yu, Xiao Jiang, and Tengjiao Wang. 2011. Relational Approach for Shortest Path Discovery over Large Graphs. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 358–369.

Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th International Conference on Experimental Algorithms (WEA'08)*. Springer-Verlag, Berlin, Heidelberg, 319–333.

Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. 2010. Fast and Accurate Estimation of Shortest Paths in Large Graphs. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM '10)*. ACM, New York, NY, USA, 499–508.

MohammadAl Hasan and MohammedJ. Zaki. 2011. A Survey of Link Prediction in Social Networks. In *Social Network Data Analytics*, Charu C. Aggarwal (Ed.). Springer US, 243–275.

Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. 2014. Hop Doubling Label Indexing for Point-to-point Distance Querying on Scale-free Networks. *VLDB Endow.* 7, 12 (Aug. 2014), 1203–1214.

Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Yu Xu. 2012a. SCARAB: Scaling Reachability Computation on Large Graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 169–180.

Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. 2012b. A Highway-centric Labeling Approach for Answering Distance Queries on Large Sparse Graphs. In *ACM SIGMOD (SIGMOD '12)*. 445–456.

Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3HOP: A High-compression Indexing Scheme for Reachability Query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 813–826.

Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. 2008. Efficiently Answering Reachability Queries on Very Large Directed Graphs. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 595–608.

Sungwon Jung and Sakti Pramanik. 2002. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE Trans. on Knowl. and Data Eng.* 14, 5 (Sept. 2002), 1029–1046.

Mehdi Kargar and Aijun An. 2011. Keyword Search in Graphs: Finding R-cliques. *Proc. VLDB Endow.* 4, 10 (July 2011), 681–692.

David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the Spread of Influence Through a Social Network. In *ACM SIGKDD (KDD '03)*. 137–146.

Paolo Massa and Paolo Avesani. 2006. Trust-aware bootstrapping of recommender systems. In *ECAI Workshop on Recommender Systems*. 29–33.

Kazuya Okamoto, Wei Chen, and Xiang-Yang Li. 2008. Ranking of Closeness Centrality for Large-Scale Social Networks. In *Frontiers in Algorithmics*. Lecture Notes in Computer Science, Vol. 5059. 186–195.

Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast Shortest Path Distance Estimation in Large Networks. In *ACM CIKM (CIKM '09)*. 867–876.

Miao Qiao, Hong Cheng, Lijun Chang, and J.X. Yu. 2014. Approximate Shortest Distance Computing: A Query-Dependent Local Landmark Scheme. *IEEE Transactions, Knowledge and Data Engineering* 26, 1 (Jan 2014), 55–68.

Michael Rice and Vassilis J. Tsotras. 2010. Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions. *Proc. VLDB Endow.* 4, 2 (Nov. 2010), 69–80.

Peter Sanders and Dominik Schultes. 2005. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Algorithms ESA 2005*, GerthStlting Brodal and Stefano Leonardi (Eds.). Lecture Notes in Computer Science, Vol. 3669. Springer Berlin Heidelberg, 568–579. DOI:http://dx.doi.org/10.1007/11561071_51

Lenie Sint and Dennis de Champeaux. 1977. An Improved Bidirectional Heuristic Search Algorithm. *J. ACM* 24, 2 (April 1977), 177–191.

Christian Sommer. 2014. Shortest-path Queries in Static Networks. *ACM Comput. Surv.* 46, 4, Article 45 (March 2014), 31 pages.

Yufei Tao, Cheng Sheng, and Jian Pei. 2011. On K-skip Shortest Paths. In *ACM SIGMOD*. 421–432.

Robert Tarjan. 1972. Depth first search and linear graph algorithms. *SIAM J. Comput.* (1972).

Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Bañuelos, Jaak Vilo, and Marlon Dumas. 2011. Fast Fully Dynamic Landmark-based Estimation of Shortest Path Distances in Very Large Graphs. In *ACM CIKM*. 1785–1794.

Antti Ukkonen, Carlos Castillo, Debora Donato, and Aristides Gionis. 2008. Searching the Wikipedia with Contextual Information. In *ACM CIKM (CIKM '08)*. 1351–1352.

Dorothea Wagner and Thomas Willhalm. 2007. Speed-Up Techniques for Shortest-Path Computations. In *STACS 2007*, Wolfgang Thomas and Pascal Weil (Eds.). Lecture Notes in Computer Science, Vol. 4393. Springer Berlin Heidelberg, 23–36. DOI: http://dx.doi.org/10.1007/978-3-540-70918-3_3

Fang Wei. 2010. TEDI: Efficient Shortest Path Query Answering on Graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 99–110.

Yang Xiang. 2014. Answering Exact Distance Queries on Real-world Graphs with Bounded Performance Guarantees. *The VLDB Journal* 23, 5 (Oct. 2014), 677–695.

Da Yan, J. Cheng, W. Ng, and S. Liu. 2013. Finding distance-preserving subgraphs in large road networks. In *ICDE*. 625–636.

Hilmi Yildirim, Vineet Chaoji, and MohammedJ. Zaki. 2012. GRAIL: a scalable index for reachability queries in very large graphs. *The VLDB Journal* 21, 4 (2012), 509–534.

Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability Queries on Large Dynamic Graphs: A Total Order Approach. In *ACM SIGMOD*. 1323–1334.

Andy Diwen Zhu, Hui Ma, Xiaokui Xiao, Siqiang Luo, Youze Tang, and Shuigeng Zhou. 2013a. Shortest Path and Distance Queries on Road Networks: Towards Bridging Theory and Practice. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. 857–868.

Andy Diwen Zhu, Xiaokui Xiao, Sibo Wang, and Wenqing Lin. 2013b. Efficient Single-source Shortest Path and Distance Queries on Large Graphs. In *ACM SIGKDD*. 998–1006.

Uri Zwick. 2001. Exact and Approximate Distances in Graphs A Survey. In *Algorithms ESA 2001*, FriedhelmMeyer auf der Heide (Ed.). Lecture Notes in Computer Science, Vol. 2161. Springer Berlin Heidelberg, 33–48. DOI: http://dx.doi.org/10.1007/3-540-44676-1_3

# Online Appendix to: TopCom: Index for Shortest Distance Query in Directed Graph

VACHIK S. DAVE, Indiana University Purdue University, Indianapolis
MOHAMMAD AL HASAN, Indiana University Purdue University, Indianapolis

This paper is an extension of the short paper published in an international conference [Dave and Hasan 2015]. In this journal version paper, we have made considerable additions to the short paper, which are listed below:

(1) We have added an important theoretical explanation with proof that our TOPCOM is the 2-Hop indexing method, which is the most accepted indexing method from the last decade. We used the well known mathematical induction technique for this proof which is explained in the section 3.5.
(2) Our short paper version included indexing for only DAG (Directed Acyclic Graph). In the journal version we have extended our indexing method for arbitrary directed graphs. The description of the method is available in section 4.
(3) In section 5 we have added experimental evaluation of the extended method for any directed graph that shows considerable improvement over state of the art methods.
(4) We have also added experimental study to compare scalability of TopCom with TreeMap. The comparison is performed over graphs with $10000, 15000, 20000$ and $25000$ nodes and their average degrees being $0.5, 1, 2, 3, 4$ and $5$. So $24$ graphs with different sizes and densities were used for checking the scalability of the indexing method. This comparative study shows that TopCom is significantly better than TreeMap for huge graphs. The description is available in section 5.2.
(5) A newly added section 2 is a related work section which has brief descriptions of the recent research on indexing.

We believe there is atleast 45-50% additional content in the journal version as compared to the short paper.


DOI: http://dx.doi.org/10.1145/0000000.0000000