Evaluating “find a path” reachability queries

Panagiotis Bouros[1] and **Theodore Dalamagas**[2] and **Spiros Skiadopoulos**[3] and **Timos Sellis**[4]

Abstract. Graphs are used for modelling complex problems in many areas, such as spatial and road networks, social networks, Semantic Web. An important type of queries in graphs are reachability queries. In this paper, we consider the problem of answering “find a path” reachability queries. Given two nodes s and t in a graph, we want to find a path from s to t. To this end, we propose a novel representation of a graph as a set of paths that preserve the reachability information and introduce $\mathcal{P}$-Index to index and provide efficient access in this representation. Then, we extend the depth-first search algorithm to work with the paths of the representation, instead of the graph edges, for evaluating “find a path” reachability queries. Finally, we conduct a preliminary set of experiments that indicate the advantage of exploiting a set of paths for efficiently answering “find a path” reachability queries instead of using the edges of the graph.

1 INTRODUCTION

Graph data management is important for several application areas. Examples include spatial networks (i.e., road systems), social networks (i.e., user communities) and web log analysis. An important type of graph queries, known as reachability queries, involve whether there is a path connecting two nodes.

The problem of evaluating reachability queries has been studied in the context of labeling schemes [1]. Each node in a graph is assigned a set of labels such that the descendants of a node can be identified easily based on node labels. These works determine whether there exists a path between two nodes, but they do not actually identify that path.

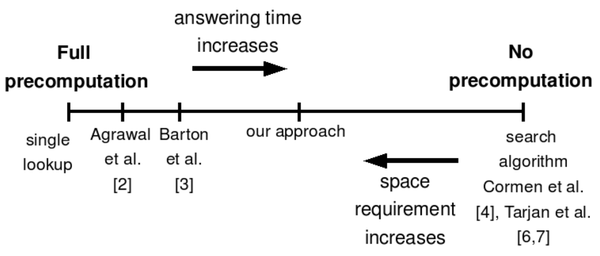

Figure 1. Approaches for evaluating “find a path” reachability queries.

[1] National Tech. Univ. of Athens, Greece, email: pbour@dblab.ece.ntua.gr
[2] Institute for the Management of Information Systems — “Athena” R.C., Greece, email: dalamag@imis.athena-innovation.gr
[3] University of Peloponnese, Greece, email: spiros@uop.gr
[4] National Tech. Univ. of Athens and Institute for the Management of Information Systems — “Athena” R.C., Greece, email: timos@imis.athena-innovation.gr

There are two approaches for identifying a path that connects two nodes in case of a reachability query. The first approach is to use a search algorithm [4] to traverse the graph until either the target node is reached or there are no other accessible nodes. The second approach is to precompute and store the path information from the transitive closure of the graph. Then, paths can be identified by just a single lookup. This second approach is clearly more efficient than the first one, where no precomputation of path information takes place. However, precomputing and storing such information costs a lot. Figure 1 depicts these two solutions in evaluating “find a path” reachability queries. The on-the-fly algorithmic solutions are located at the right “no precomputation” end of the line. Near the “full precomputation” end there exist approaches that try to deal with the cost of computing and storing path information from the transitive closure of the graph, either exploiting encoding schemes or graph segmentation (see related work in Section 2).

In this paper, we deal with the evaluation of “find a path” reachability queries: given two nodes s and t in a graph, find a path from s to t, if any. We introduce a novel graph representation, called path representation. Given a graph $G(V, E)$, the idea is to have a set of paths that we can use in order to form G. Using this representation, we actually precompute and store part of the complete path information from the transitive closure of the graph, so that we can exploit this information for query evaluation. To efficiently access path representations, we maintain an inverted index called $\mathcal{P}$-Index. Exploiting path representation and its $\mathcal{P}$-Index, we extend the depth-first search algorithm to operate on the paths of the representation instead of the edges of the graph in order to evaluate “find a path” reachability queries efficiently. Compared to the approaches presented in Figure 1, our approach is in the middle of the full-no precomputation range, since we neither have to compute the transitive closure of the graph nor have to store or encode all the paths between any two nodes.

Contribution. The key contributions of this paper are:

- We propose a novel representation of a graph, called path representation, to store path information of the graph.
- We introduce $\mathcal{P}$-Index to index and provide efficient access in path representations.
- We extend depth-first search algorithm to exploit path representation for answering “find a path ” reachability queries efficiently.
- We present a preliminary experimental evaluation to show the advantages of our approach.

Outline. First, in Section 2 we review literature on evaluating “find a path” reachability queries. Then, Section 3 presents our notation for graph-theoretic terms. In Section 4 we introduce the path representation of a graph and present an index for it. In Section 5 we discuss an algorithm for answering “find a path” reachability queries exploiting a path representation. Our experimental findings are presented in

Section 6. Finally, Section 7 concludes the paper.

2 RELATED WORK

One way to evaluate “find a path” reachability queries on graphs is to traverse the graph at query time using a depth-or breadth-first search algorithm [4]. Another approach is Tarjan’s solution to single source path expression problem from [6, 7].

In [2] the authors propose an encoding scheme for storing a semi-materialized view of the path information from the transitive closure of a graph. The work lies close to the “full precomputation” end in Figure 1 since it computes all possible paths between any two graph nodes but it does not actually stores them. The encoding scheme assigns to each node v a set of triples $\langle destination, via, label\rangle$, where “via” denotes the first hop in the path from the v to the destination node. Thus, for each node that is accessible from v they create the corresponding triple. At query time, they answer “find a path” reachability queries by performing a number of lookups on their encoding scheme. Yet, the creation of the encoding scheme still requires to compute the transitive closure of the graph. In [3] Bartoň and Zezula provide a different approach in storing path information. They introduce ρ-index: a multilevel balanced tree structure where each node is a graph segment created by a graph segmentation procedure. For each node of ρ-index (segment of the graph), they compute its transitive closure and store its complete path information. Compared to [2], this work does not compute or store the path information from the transitive closure of the whole graph but only from its segments and therefore is located less close to the “full precomputation” end of Figure 1. Yet, the search algorithm can only find paths between two nodes of length at most equal to a specific threshold, which is a construction parameter of the index.

In the context of labeling schemes, [1] proposes an interval labeling scheme for DAGs. The first step is to compute the spanning tree of the graph and then considering also the graphs edges excluded from the spanning tree, they assign to each node v a sequence $L(v)$ of intervals. In [9], Wang et al. introduce Dual-Labeling for sparse DAGs. They also compute the spanning tree of the graph and then compute the transitive closure of the graph edges outside the spanning tree. Each node is assigned two labels: one according to the spanning tree edges and another label for the rest of the edges. In [8] Trißl et al. introduce GRIPP scheme for large DAGs. They also assign to each node an interval label but unlike the previous works, they do not compute the spanning tree of the DAG.

3 PRELIMINARIES

We begin by presenting our notation and some graph-theoretic terms and properties.

Definition 3.1 (Graph) *A* directed graph*, or simply a* graph *G, is a pair (V, E), where V is a set of* nodes *and $E \subseteq V \times V$ is a set of ordered node pairs (v_i, v_j) called* edges.

Definition 3.2 (Path) *Let $G(V, E)$ be a graph. A* path *$p(v_1, \ldots, v_k)$ is a sequence of distinct nodes $(v_1, \ldots, v_k) \in V$ such that $(v_i, v_{i+1}) \in E$, for each $i \in [1, k)$. By* nodes(p) *and* edges(p) *we denote the set of nodes and the set of edges in p respectively.*

Example 3.1 Figure 2 presents an example of a graph G. $p(A, B, C, E)$ is a path in G, with $nodes(p) = \{A, B, C, E\}$ and $edges(p) = \{(A, B), (B, C), (C, E)\}$.

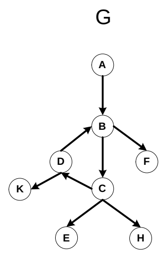

Figure 2. An example of a graph.

We can represent a graph by the set of its edges.

Definition 3.3 (Edge representation) *The* edge representation *of a graph $G(V, E)$, denoted by $\mathcal{E}(G)$, is the set of its edges E.*

Adjacency list can be used to index the edge representation of a graph.

Definition 3.4 (Adjacency list) *Let $G(V, E)$ be a graph and $\mathcal{E}(G)$ be its edge representation. For each node v of G, the* adjacency node list *of v, denoted by $adj[v]$, is the list of nodes u such that $(v, u) \in \mathcal{E}(G)$, i.e., $(\forall v \in V)(adj[v] = \{u|(v, u) \in \mathcal{E}(G)\})$. The* adjacency list *of G, denoted by $ADJ(G)$, is the set of the adjacency node lists of all nodes in G, i.e., $(ADJ(G) = \{adj[v] \mid \forall v \in V\})$.*

Example 3.2 Table 1 illustrates the adjacency list of graph G in Figure 2.

Table 1. Adjacency list of graph G in Figure 2.

node	adjacent nodes
A	B
B	C, F
C	D, E, H
D	B, K
E	
F	
H	
K	

4 PATH REPRESENTATION OF A GRAPH

In this section, we introduce a novel representation of a graph, called *path representation*, to store part of the path information of the graph. The key idea is to combine the edges of the graph for defining a set of paths that can be used to efficiently answer “find a path” reachability queries. To preserve the reachability information of a given graph, its path representation includes all its edges. In other words for each edge (v, u) of the graph there is a path in the path representation that includes (v, u).

Definition 4.1 (Path representation) *The* path representation *of a graph $G(V, E)$, denoted by $\mathcal{P}(G)$, is a set of distinct paths $P = \{p_1, \ldots p_m\}$ such that $\bigcup_{i=1}^{m} edges(p_i) = E$, i.e., G can be constructed by merging all the paths in P.*

Note that, contrary to the edge representation of a graph, path representation is not unique. A given graph may be represented using several path representations.

Example 4.1 Consider graph G of Figure 2. Table 2 depicts a path representation of G: $\mathcal{P}(G) = \{p_1, p_2, p_3, p_4\}$.

Table 2. A path representation of graph G in Figure 2.

p_1	(A, B, C, E)
p_2	(C, D, B, F)
p_3	(C, H)
p_4	(D, K)

Next, we introduce *$\mathcal{P}$-Index* to index a path representation. In short, for each node v of a graph G, $\mathcal{P}$-Index stores a list of the paths in $\mathcal{P}(G)$ that contain v.

Definition 4.2 ($\mathcal{P}$-Index) *Let $G(V, E)$ be a graph and $\mathcal{P}(G)$ be one of its path representations. For each node v of G, the* path list *of v, denoted by $paths[v]$ is the list of paths $p \in \mathcal{P}(G)$ that contain v, i.e., $(\forall v \in V)(paths[v] = \{p | p \in \mathcal{P}(G) \wedge v \in nodes(p)\})$. The* path index *of G, denoted by $\mathcal{P}$-Index(G), is the set of the path lists of all nodes in G, i.e., ($\mathcal{P}$-Index$(G) = \{paths[v] \mid \forall v \in V\}$)*

Example 4.2 Consider graph G in Figure 2 and its path representation in Table 2. Table 3 illustrates the $\mathcal{P}$-Index for G.

Table 3. $\mathcal{P}$-Index for the path representation in Table 2 of graph G in Figure 2.

node	paths containing node
A	p_1
B	p_1, p_2
C	p_1, p_2, p_3
D	p_2, p_4
E	p_1
F	p_2
H	p_3
K	p_4

5 EVALUATING "FIND A PATH" REACHABILITY QUERY

In this section, we present our method for answering "find a path" reachability queries exploiting a path representation of a graph.

Definition 5.1 ("find a path" reachability query) *Let $G(V, E)$ be a graph and s, t be two of its nodes. A* "find a path" reachability query *in G, denoted by* `FindAPath`(s, t), *identifies a path starting from node s and ending at t.*

Next, we present Algorithm pdfs that exploits $\mathcal{P}$-Index to answer `FindAPath` queries. In summary, Algorithm pdfs:

- extends the depth-first search procedure, visiting for a node v, not only its adjacent nodes, but also the ones following it in the path representation, and
- exploits $\mathcal{P}$-Index to determine efficiently whether there exists a path in the representation containing a node v and the target node.

Figure 3 illustrates the pseudo-code of Algorithm pdfs. Initially, pdfs checks whether both the source s and the target node t are contained in a path r of the representation with s before t (Lines 2-3). If so, then an answer path is found. Otherwise, it creates the stack $curPath$ to store, in each iteration of the search, the path from the source node s to the current search node. In each iteration, the algorithm gets the next path p containing the top node u of the $curPath$ stack (Lines 7-8). Node u is considered the current search node. Then, it visits each node v that lies after u in path p (Lines 12-19) until it reaches an already visited node (Line 18) or there is a path r in the representation that contains first v and then the target node t (Lines 14-15). The algorithm pops a node from $curPath$ after having read all the paths that contain it (Lines 9-10).

To identify a path in the representation that contains a given node v before the target t, pdfs algorithm joins the corresponding $paths[v]$ and $paths[t]$ lists of $\mathcal{P}$-Index. The $paths$ lists in $\mathcal{P}$-Index are sorted by the path identifier and therefore the algorithm actually performs a merge-join. The join procedure terminates when a common path r that contains node v before t is found or one of the $paths$ list is traversed till its end.

```
Algorithm pdfs
Input: graph $G(V, E)$, path representation $\mathcal{P}(G)$, $\mathcal{P}$-Index$(G)$,
       nodes $s$ and $t$
Output: a path from $s$ to $t$
1  begin
2      if exists path $r \in \mathcal{P}(G)$ containing $s$ before $t$ then
3          return SubPath($r, s, t$);                    // found path
4      create stack $curPath$ containing source node $s$;
5      mark $s$ as visited;
6      while $curPath$ is not empty do
7          read the top node $u$ of $curPath$;
8          get the next path $p$ in $\mathcal{P}(G)$ containing $u$;
9          if have read all paths containing $u$ then
10             pop $u$ from $curPath$;
                   // checked all possible paths through $u$
11         else
12             foreach node $v$ in $p$ after $u$ do
13                 if $v$ is not visited then
14                     if exists path $r \in \mathcal{P}(G)$ containing $v$ before
                       $t$ then
15                         return $curPath \cup$ SubPath($r, v, t$);
                                                         // found path
16                     push $v$ in $curPath$;
17                     mark $v$ as visited;
18                 else if $v$ is visited then
19                     ignore rest of nodes in $p$;
                                                  // avoid graph cycle
20     return null;
21 end
```

Figure 3. Algorithm pdfs

Example 5.1 Consider graph G in Figure 2, its path representation $\mathcal{P}(G)$ in Table 2, $\mathcal{P}$-Index(G) in Table 3 and `FindAPath`(B, K) query. There exists no path in $\mathcal{P}(G)$ containing first source node B

and then target node K. Therefore, pdfs creates stack $curPath$ with the source node B. $p_1(A, B, C, E)$ is the first path containing B. The algorithm checks whether there exists a path in $\mathcal{P}(G)$ containing C before K or E before K and afterwards it inserts nodes C and E in $curPath$. The next node to consider is E. There exists no path in $\mathcal{P}(G)$ containing E before K. Moreover, E is contained only in p_1 and in fact at the end of the path. Thus, the algorithm removes node E from $curPath$, since there is no way to reach target K through E, and backtracks to C. For C, p_1 can not be used because the successor node E is already visited. Therefore the algorithm reads the next path containing C, i.e., $p_2(C, D, B, F)$. Before visiting D, pdfs identifies that path p_4 contains node D and then target K. Thus, the answer of `FindAPath`(B, K) is the $curPath$ followed by the subpath of p_4 from D to K, i.e., (B, C, D, K).

6 EXPERIMENTS

We present a preliminary experimental evaluation of our approach. Our intention is to demonstrate the advantage of exploiting a path representation of a graph, indexed by $\mathcal{P}$-Index, over using the edge representation of the graph, indexed by its adjacency list, for answering `FindAPath` queries. For the latter, we have implemented a standard depth-first algorithm (called dfs) that uses the adjacency list to traverse a graph until a path between two given nodes is found or there are no more accessible nodes. Both algorithms are implemented in C++, compiled with gcc and executed on a 3 Ghz Intel Core 2 Duo CPU.

For our experiments, we generate random graphs exploiting the NetworkX library [5] and we construct their path representations. For the latter, we traverse the graph in a depth-first manner, starting from several random nodes. We terminate the procedure when all graph edges are included in some path. We promote the construction of long against that of short paths by using edges of the graph more than once. In this way pdfs algorithm visits more nodes in each iteration and in addition it is more possible to identify a path in the path representation of the graph that contains a specific node before the target one.

We identify three experimental parameters: (a) the number of nodes $|V|$ and (b) the average degree $d = |E|/|V|$ of the graph, and (c) the maximum length L_{max} of the paths in the path representation. The graphs of our dataset have from $10,000$ to $1,000,000$ nodes and from 2 to 10 average degree. The maximum length of the paths in the representations varies from 10 to 50. Table 4 summarizes the parameters involved, their ranges and their default values. We perform three kinds of experiments. In each experiment we vary a single parameter while we set the remaining ones to their default values.

Table 4. Experimental parameters

parameter	values	default value
L_{max}	10, 20, 30, 40, 50	30
$\|V\|$	10^4, $5 \cdot 10^4$, 10^5, $5 \cdot 10^5$, 10^6	10^5
$d = \|E\|/\|V\|$	2, 3, 4, 5, 10	4

Varying the maximum length of the paths in a representation. First, we study the effect of the maximum length of the paths in answering `FindAPath` queries. For a graph of $100,000$ nodes and an average degree of 4, we construct 5 different representations by varying the maximum length of their contained paths. Figure 4 shows the average execution time for answering one of the $1,000$ `FindAPath` queries posed. The results indicate that the average execution time decreases as L_{max} grows. For example pdfs needs 1.5 secs in average to answer a `FindAPath` query exploiting the representation of max length 10, whereas in case of the representation $L_{max} = 40$ the time drops at less than 0.5 sec. The main reason for this behavior is that, as the maximum length of the paths increases, the representations include larger part of the path information of the graph. In addition, as L_{max} increases, the representations contain fewer and longer paths with more common nodes since they combine more edges with each other. Therefore it is more possible to identify a path that contains a specific node before target one.

On the other hand, as the maximum length of the paths increases and the representations include larger part of the path information of the graph, the space needed to store the representations grows. Figure 5 shows the overhead in storing the path representations as L_{max} increases. For example, the presentations of maximum length 10 contains 2.4 times more edges in total than the edge representation of the graph has, whereas the one with $L_{max} = 30$ has 3.5 times more.

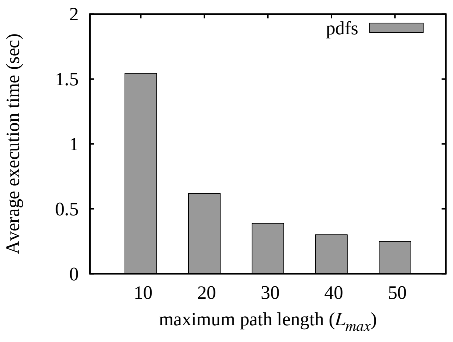

Figure 4. Average execution time varying L_{max}, $|V|$ fixed at 10^5, d fixed at 4.

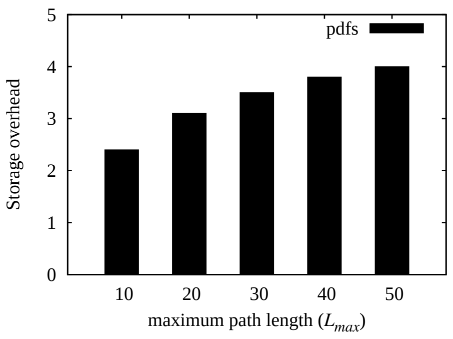

Figure 5. Storage overhead in path representations varying L_{max}, $|V|$ fixed at 10^5, d fixed at 4.

Varying the average degree. Next, we study the impact of the average degree of a graph. We construct an initial graph of $100,000$ nodes and of average degree 2 and progressively insert new edges between the existing nodes to create 4 more graphs of average degree 3, 4, 5 and 10. The path representations created for each of these graphs contain paths of length at most equal to 30. Then, we generate $1,000$ random FindAPath queries on the initial graph and evaluate them over all 5 graphs. Figure 6 presents the average execution time per query. The execution time of dfs increases for more dense graphs. In contrast, pdfs not only outperforms dfs in any case but also its execution time goes down as d increases. For example dfs needs 3.5 secs to answer a query on the initial graph, but for the more dense graph of average degree 5 it needs almost 5.5 secs. On the other hand, it takes 2.5 secs in average for pdfs to evaluate FindAPath queries on the initial graph, but less than 0.5 sec on the graph of average degree 5. This is due to the fact that, for more dense graphs, the number of the long paths in the representation increases whereas the number of the short ones decreases.

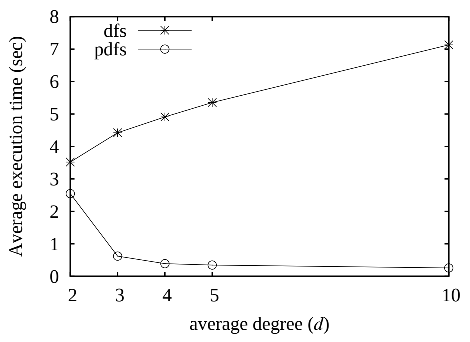

Figure 6. Average execution time varying average degree d, $|V|$ fixed at 10^5 and L_{max} fixed at 30.

Varying the number of nodes. Finally, we test how the number of graph nodes affects the average execution time of FindAPath queries. We create 5 graphs with average degree 4 and pose $1,000$ FindAPath queries on each. Figure 7 presents the average execution time per query. Both dfs and pdfs are affected by the increase of nodes. Their execution time goes up as the number of nodes $|V|$ increases. However, the impact on pdfs is less intense and moreover its average execution time is lower. For example it takes over 2.5 secs for dfs to evaluate a query on a graph of $50,000$ nodes and near 35 secs on a larger one of $500,000$ nodes. In contrast, pdfs needs less than 0.5 sec for the $50,000$ nodes graph and almost 3 secs for the $500,000$ one. Note that, for graphs with larger set of nodes and fixed average degree, the paths in the representations have fewer common nodes and thus, it is less possible to identify a path in the representation that contains a specific node before target one. The advantage of pdfs is presented also in Figure 8. The average number of nodes processed per query by pdfs is far less than in case of dfs.

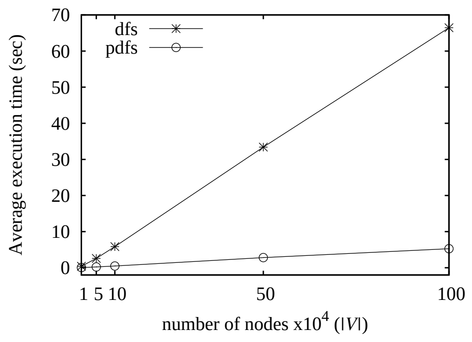

Figure 7. Average execution time varying $|V|$, d fixed at 4 and L_{max} fixed at 30.

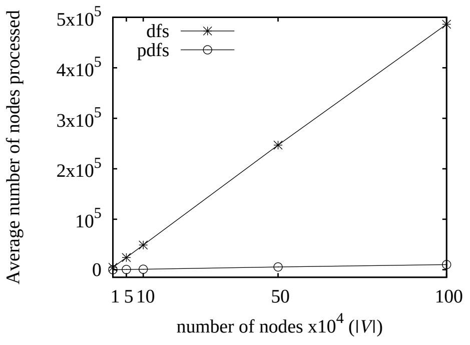

Figure 8. Average number of nodes process varying $|V|$, d fixed at 4 and L_{max} fixed at 30.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we consider the problem of evaluating “find a path” reachability queries on graphs. We represent a graph as a set of paths, called path representation, that include part of the path information from the transitive closure of the graph and present P-index to index and provide efficient access in path representations. Then, an extended depth-first search algorithm that operates on paths exploits the representation to efficiently answer “find a path” reachability queries between any two graph nodes. Our experiments indicate the advantage of considering a path representation instead of using the edges of the graph. In the future, we plan to further investigate the construction of the path representations of a graph and to deal with their incremental updates. In addition, we will apply the idea of exploiting paths to answer other kinds of queries, e.g. finding shortest or critical paths.

REFERENCES

[1] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish, ‘Efficient management of transitive relationships in large data and knowledge bases’, in *SIGMOD Conference*, pp. 253–262, (1989).

[2] Rakesh Agrawal and H. V. Jagadish, ‘Materialization and incremental update of path information’, in *ICDE*, pp. 374–383, (1989).

[3] Stanislav Bartoň and Pavel Zezula, ‘Designing and evaluating an index for graph structured data’, in *ICDM Workshops*, pp. 253–257, (2006).

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Second Edition*, The MIT Press and McGraw-Hill Book Company, 2001.

[5] NetworkX, High productivity software for complex networks. https://networkx.lanl.gov/.

[6] Robert Endre Tarjan, ‘Fast algorithms for solving path problems’, *J. ACM*, **28**(3), 594–614, (1981).

[7] Robert Endre Tarjan, ‘A unified approach to path problems’, *J. ACM*, **28**(3), 577–593, (1981).

[8] Silke Trißl and Ulf Leser, ‘Fast and practical indexing and querying of very large graphs’, in *SIGMOD Conference*, pp. 845–856, (2007).

[9] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu, ‘Dual labeling: Answering graph reachability queries in constant time’, in *ICDE*, p. 75, (2006).